Microsoft® Language シリーズ

7

Quick BASIC
中級プログラミング

小池慎一＋森 博［著］

技術評論社

Microsoft® Language シリーズ

7

Quick BASIC
中級プログラミング

小池慎一＋森 博［著］

技術評論社

まえがき

プログラミングには，知的なパズルといった面があります．

学習者や非専門家プログラマにとっては，時間さえあればどんなプログラムでも書けるという信念があります．キーボードに向かっていると次々とアイデアがわいてきて，知らぬまに“素晴らしい”プログラムができていたという経験は誰にでもあることでしょう．

はじめてプログラムを始める人，とくに独習者にとって，BASIC言語はまことに良い環境を提供してくれました．対話型で，“呼べば応える”コンピュータは，知らぬまに手足のごとく働く道具となりました．

時とともにハードが進歩したばかりでなく，BASIC自体も進歩し，ついにQuick BASICが登場しました．それにより，それまでの制約のいくつかが克服され，プログラミング環境は一層充実したものになりました．

本書を書く動機となったのは，Quick BASICを知るために実際にユーティリティプログラムをいくつも作成してみた経験です．プログラミングを通じて，いくつかの制約にぶつかりはしましたが，基本的にはこれならば充分に実用になること，アルゴリズムの記述にも適しており，学習用にも勧められることが明らかになりました．そこで，ぜひ多くの同好の士に紹介したいと考えました．

構成としては，プログラミングの基礎となる，データ構造とアルゴリズムについての解説を前半にまとめ，後半にプログラミングの例を説明することにしました．

執筆は，小池と森が次のように分担しました．

はじめに，１．１，１．５～１．７，２．５～２．７　　小池

１．２～１．４，２．１～２．４　　森

最後に，構想から完成まで１年余りもかかり，いろいろご迷惑をかけましたが，それにもかかわらず出版にいたりましたことを，編集部の加藤氏，㈱パピルスの高松さんに感謝いたします．

1990年８月　　著　者

目次

第2部　Quick BASICプログラム研究 ……………………101

はじめに
Quick BASIC
誕生までの足どり

Quick BASIC

パーソナルコンピュータにおけるBASIC発展の足どりは，マイクロソフト社のBASIC発展のそれでもあります．現在までいろいろなBASICが誕生してきましたが，生き延びているのは，数の上でもほとんどありません(日本では，BASIC/98が頑張ってはいますが)．

BASICは，よく知られているように，TSS(Time Sharing System)環境下でプログラミング教育をする目的で，1965年，ダートマス大学において，J. D. ケメニーとT. E. カーツのもとで開発されました．その後1975年時点では，FORTRANにつぐ人気のある言語となりました．これらは，いずれもTSSのもとで動く対話型言語でした．

一方，1971年にインテル社により，はじめてのマイクロプロセッサが開発されました．1975年には，ミッツ社がマイクロコンピュータを開発し，その上で動く4Kバイトと8Kバイトの大きさのBASICを発表しました．このBASICを開発したのが，B. ゲーツとP. アレンで，この2人はのちのマイクロソフト社の創立者となりました．

一方，アマチュアの間でも，マイクロプロセッサを使用して小さなマイクロコンピュータの製作が行われました．当時はROMもRAMも高価であり，技術的な理由からも，小さなメモリしか持っていませんでした．そこで，2KバイトのROMの上にも載るTINY(小さな)BASICが開発されました．わが国でも，東大が中心になって全国のホビーイストや大学・研究機関に広めました．

1977-78年にかけて，パーソナルコンピュータと呼べる3種のマシン，PET，Apple II，TRS-80がそれぞれ，コモドール，アップル，タンディラジオシャック社から発表されました．それらは，いずれも8KBASICに拡張機能を加え合わせた本格的なBASICを搭載していました．

話は前後しますが，8KBASICは，現在のBASICの原型となったもので，ABS，ATN，COS，SIN，EXP，LOG，SQR，RNDなどの数学関数，LEFT$，RIGHT$，MID$などの文字列関数，ASC，CHR$，STR$，VALなどの型変換関数を持ち，それらはQuick BASIC (以下QBと略す) にもそのまま残っています．また，24ビットの単精度数や16ビットの整数も引き継がれています．1行のIF/THEN/ELSEも残されています．QBでは使用頻度は減少したとは言え，GOSUB/RETURNも健在です．のちに強化されることになる，グラフィックス，サウンド，ファイル処理を別にすれば，8KBASICの骨格は，その後ずっと変化していないと

言えます.

マイクロソフト社のMicrosoft BASIC（以下MBASICと略す）は，OS CP/M上で動くものがMBASIC 4，そのあとに改良されたバージョンがMBASIC 5と呼ばれます.日本で広く用いられているN_{88}-BASICの基本的な機能はこのMBASIC 5に同等と考えられます.

BASICはQBが登場するまでは,インタプリタであって,対話型で使用できるので使いよいが，速度は遅いと言うのが定説でした．実行速度が遅いことに対してはBASICコンパイラBASCOMで対応しましたが,使い勝手の悪さは否定できませんでした.

対話型に近い環境とコンパイル速度の速さを売りものにしたTURBO Pascalが登場したのは，MBASIC 4の時代で，日本でも1985年頃から普及してきました. 8ビットのCP/Mの時代でも,高速性にひかれてTURBO Pascalを使用したものですが,16ビットのMS-DOSの時代になり,まともな仕事に使うのならばTURBO Pascalという時代になりました．それに対して，マイクロソフト社は，BASICの改良につとめ,TURRO Pascalの新しいバージョン4.0以降のバージョンに対抗可能なQuick BASICを発表しました．統合環境版と呼ばれる操作方法はTURBO Pascalのそれにきわめてよく似ており，コンパイラでありながら，インタプリタ並みのデバッグしやすさを実現しています．能力的にもPascalに匹敵するものを持ち,Pascalに流れたユーザを引き戻すこともできるくらいになったと言えます.

競争は進化の源と言われますが，TURBO Pascalとの競争がこの優れたQuick BASICを生み出したことに感謝すべきでしょう（逆に言えば，多くの人がBASICの不備――すべてが1個のブロックでありローカル変数が持てないこと，複数行のIF/THEN/ELSEが不可能なことなど――を雑誌等で指摘していたにもかかわらず，また，有効桁数が7桁ありながら印刷上は6桁に丸めてしまうことなどに対しても，マイクロソフト社は何の手も打たなかったのですから，もし，TURBO Pascalを作ったボーランド社というライバルが出現しなければ,BASICはまだずっとほおっておかれたのかもしれません).

以上のように書くと，BASICの歴史は良くも悪くもマイクロソフト社のそれのように思われますが，マイクロソフト社以外のアメリカの各社や，ヨーロッパやソ連では独自の発達をしています．たとえば，BASIC/F（ソ連）では，モジュールやQBのDO/LOOPに相当するループ構造はすでに実現されています.同様に，ECMA

BASIC(ヨーロッパ)にはDO/LOOPが，BetterBASIC(アメリカ)にはDO/END . . . DOやDO/REPEATなどがあります．

また，MBASICの“恐ろしい”FIELD文を使用しないファイルアクセスは，前出のBASIC/F，ECMA BASICのほかにCBASICやRSBASIC(共にアメリカ)などでは早くから実現されています．

ただ，ヨーロッパやソ連のBASICは日本には入ってきませんし，マイクロソフト社以外の各社のBASICは大きな市場を持つことができなかったために，TURBO Pascalのようにマイクロソフト社の脅威とはなり得なかったと言えます．良いものが，市場を支配するとはかぎらない例です．

参考文献

(1) 小池慎一「マイコンとBASIC」マイコンピュータ　CQ出版社　No. 13, p2-11(1984)

(2) A. E. コルチャク，小池訳「BASIC言語比較研究——BASIC言語の本質とパソコンの歴史」技術評論社(1989)

(1)はマイコン登場から1983年頃までの，日米のBASICについてまとめられている．

(2)はアメリカ，ヨーロッパ，ソ連にわたって，パーソナルコンピュータ用BASICの発展についてまとめられている．

第1部
データ構造とアルゴリズム

1.1 Quick BASICの特徴と留意点

Quick BASIC(以下QBと略す)は，従来のMBASICの単なる拡張版ではなく，データ構造や制御構文の点でPascalに匹敵するほどの機能を含んでいます．BASICの使いにくさ，スピードの遅さから，PascalやCへ移った人を再びBASICへ呼び戻すだけのものはあります．

しかし，QBはBASICの仕様を引きずっているため，せっかくここまでやったのだからもう少し改良してほしかったと思われる点も多くあります．また，PascalやCに似ていますが異なる点もあります．そこで，本章ではQBを使用するうえでの留意事項についてまとめてみます．

1.1.1 宣言なしの変数の使用

BASICの特徴のひとつは，宣言なしで変数が使用できることです．言い換えると，新しい変数の登場がすなわち変数の宣言ということです．この性質は，さほど大きくないプログラムを作成する場合には便利ですが，変数名のタイプミスをチェックできないこと(タイプミスは新しい変数の宣言を引き起こす)，後述のように，変数の型についてのプログラマの注意がおろそかになることを引き起こします．プロシージャの引数の型が厳密にチェックされるのに注意してプログラミングしていると，宣言なしの変数の使用には違和感を覚えます．

また，プロシージャ内でローカル変数を用いたQBのプログラムを，PascalやCに移植する場合は次の点に注意する必要があります．プロシージャ内でローカル変数の宣言がなされていないので，グローバルな変数と同名のものが用いられていると，移植先ではローカルではなくなってしまいます．Pascalに似ているからと言っても，うっかりしていると，発見の難しいバグとなります．

QBでは，変数の宣言がすべてDIMで始まります．これは，DIMがDimensionの

略とすると，抵抗があります．BASICでの変数の宣言が従来はDIMしかなかったので，それを流用したとも考えられますが，コマンドの名前はもともとの語の持つ意味からそれるのは好ましくありません(リスト1.1.1)．

●リスト 1.1.1　DIMによる宣言の例

```
DIM S1 AS STRING * 10
DIM SHARED DataArray(DataNumbers) AS INTEGER
DIM Xpos AS Ptr
```

1.1.2 1行がひとつの文

BASICでは，文は1行で書かれます(マルチステートメントもありますが，ここではとりあげません)．したがって，改行は新しい文を意味します．

PascalやCで書かれたプログラムをQBに移す場合，QBでは文を1行に書かなければならないので困ることがあります．たとえば，代入文の右辺が長い式の場合，あるいは式そのものは長くなくても，わかりやすい変数名を使用するために文字数が多くなると，1行が長くなって読みにくくなります．IF文の条件が長い式になる場合も同様です．継続行の使用が可能になるような方法がとれるとよいと思いますが(リスト1.1.2)．

●リスト 1.1.2　文は1行に書かねばならない

```
StandardDev = SQR((SumOfSquared - SumOfData ^ 2/NumberOfData)/(NumberOfData - 1))
```

1.1.3 実数と整数

BASICでは，多くの場面で実数は整数と同様に使えます．たとえば，配列の添字に実数を使用することはなんら差し支えありません．代入の場合も，実数型の変数が整数値をとっていれば，型変換の関数を使用することなく，代入可能です．

しかし，プロシージャの引数がからむと，型チェックが行われるため，区別をする必要があります．従来のBASICの感覚でコーディングしていると，コンパイ

ルエラーで手こずることがあります．

たとえばリスト1.1.3では，プロシージャPrintArrayの２番目のパラメータiは単精度実数型です．一方，呼び出し時の引数 x(i).nextは，TYPE宣言で，next AS INTEGERと整数型になっています．すると，当然コンパイル時にエラーとなります．もちろん，プロシージャを使用せず，直接

```
PRINT Y(x(i).next)
```

とすれば，エラーとはならないので，この種の誤りはコンパイル時まで気づかず，やっかいです．ゆるやかな規則と，厳密な規則が同居しているための問題と言えましょう(リスト1.1.3)．

言い換えると，これは，配列の添字を整数型に限定すると初心者にはわかりにくいであろうとして，それをゆるめたBASICの文法をそのまま引きずっているためと言えましょう．

●リスト 1.1.3　引数の型チェックに引っかかりやすい例

```
メインモジュール
  TYPE node
      body AS STRING * 100
      next AS INTEGER
  END TYPE
  DIM x(100) AS node
  ...
  ...

   CALL PrintArray( y(), x(i).next)

 SUB プロシージャ

 SUB PrintArray( x(),i)
          RPINT x(i)
 END SUB
```

1.1.4 固定長文字列

固定長文字列は，プロシージャの引数として使用できないほかにも，いくつかの注意事項があります．

固定長文字列変数の初期値は，文字コードゼロが長さだけ並んだものです．A$のような可変長文字列の初期値がヌルであるのとは異なります．

比較する場合には，長さが一致するものでないと意味をなさないので，注意する必要があります．たとえば，

```
DIM S AS STRING * 5
S = "ABCD"
X$ = "ABCD"
```

において，SとX$とは等しくありません．S = "ABCDE" と X$ = "ABCDE"のように，長さまで一致すれば，等しくなります．固定長文字列変数に固定長文字列の長さより短い文字列を代入すると，あとは空白コード(=32)で埋められます．

固定長文字列は，ヌルをとれない点にも注意が必要です．PascalやCの文字型のつもりで，

```
DIM ch AS STRING * 1
```

と宣言された変数chはつねに長さ1を持ちます．ヌル文字列の代入はchに空白を代入します．ヌルではありません．

また，DIMで宣言された固定文字列変数は，うしろに記号$をつけても可変長文字列変数にはなりません．たとえば，

```
DIM S AS STRING * 5
```

で宣言されたSがあるとします．すると代入，

```
S$="A"
```

によって，S$は長さ1でなく5をとります．すなわち，SとS$は同一のものです．このことは，

```
X%=3
X=4.5
```

によってX%とXが区別され，別々の変数とみなされるBASICの仕様を引き継いでいるQBの文法としては，不一致を感じさせます．

1.1.5 用　語

QBは，BASICの拡張されたものであり，PascalやCの特徴も取り込んでいるので，用語はよく似ています．しかし，QB独自の使い方もあるので，類推で使用すると正しく意味が伝わらないこともあります．

1.1.5.1 パラメータと引数

用語パラメータと引数は，常識的には同じ内容のものですが，QBでは次のように使い分けています．

パラメータ……プロシージャ宣言時に与えられるプロシージャに渡したり結果を受け取る変数．普通の表現では仮パラメータ，仮引数あるいは形式パラメータなどと呼ばれるもの．

引数……………プロシージャ呼び出し時に値を渡したり結果を受け取る変数．普通の表現では，実パラメータ，実引数と呼ばれるもの．

1.1.5.2 プロシージャ

QBでのプロシージャは，SUBで宣言されるサブプログラムとFUNCTIONで宣言される関数の両方を指します．プロシージャは狭い意味ではモジュールとも呼ばれますが，QBではモジュールを別の意味に用いるので，混同しないようにします．プロシージャ１個でもモジュールになり得ます．

Pascalで言うプロシージャは手続きとも呼ばれ，QBのSUBプロシージャに相当します．一方関数はどちらも関数です．

Cは，すべてが関数ですが，voidを付して宣言されたものがQBのSUBプロシージャに相当します．Cの関数＝QBのプロシージャ（ただし，main関数を除く）です．

1.1.5.3 モジュール

QBで言うモジュールとは，プロシージャを１個以上含むソースファイルを意味します．小さなプログラムは，通常１個のファイルで構成されるので，１個のモジュールからなります．汎用のプロシージャを集めたファイルを，ファイルコマ

ンドの“サブファイル”として常駐させた場合には，そのプログラムは複数のモジュールからなります．

DIM，CONST，DIM SHARED，COMMON SHAREDなどはモジュール内でのみ有効です．モジュール間で変数を共有するためには，プロシージャの引数として渡すか，COMMONを用います．

1.1.5.4 サブルーチン

サブルーチンという用語は，QBでは，GOSUBで呼ばれるプログラムのことを指します．SUBまたはFUNCTIONで宣言されるプロシージャとは区別されます．

1.1.6 プロシージャ内で使用できないステートメント

DATAステートメントはプロシージャ内では使用できません．モジュールレベルコード部にのみ置くことができます．同一のモジュール内では，READを用いて，どのプロシージャ内からでもDATAを読み込むことが可能です．モジュール間では読む込むことはできません．

OPTION BASEステートメントは使用できません．モジュールレベルコード内で1回のみ使用できます．

1.1.7 静的配列と動的配列

QBでの配列は，コンパイル時に必要な領域が確保されるものと，実行時にDIMあるいは暗黙の宣言で現れたときに領域が確保されるものの2種類があります．前者を静的($STATIC)配列，後者を動的($DYNAMIC)配列と呼びます．

ステートメントERASEは，N_{88}-BASICなどの場合と働き方が異なります．ERASEは静的配列に対しては配列の初期化を，動的配列に対しては領域の解放を実行します．

配列の静的および動的な宣言は，メタコマンド'$STATICと'$DYNAMICでなされます．リスト1.1.4に例を示します．

●リスト 1.1.4　静的配列と動的配列の宣言の例

```
'$STATIC

    PRINT FRE(0),FRE(-1)
    DIM X(100),A$(100)        'Xと A$は 静 的 配 列
    PRINT FRE(0),FRE(-1)

'$DYNAMIC
    DIM Y(100),B$(100)        'Yと B$は 動 的 配 列
    PRINT FRE(0),FRE(-1)
    ERASE X,Y,A$,B$
    PRINT FRE(0),FRE(-1)

出 力 の 例
   41806     129438
   41806     129438      <--  Xと A$は 静 的 な の で 、 は じ め の 状 態 に 等 し い
   41394     128610      <--  Yと B$の 宣 言 で メ モ リ が 消 費 さ れ た
   41806     129438      <---  ERASEに よ っ て Yと B$の 領 域 が 解 放 さ れ た
```

一方，配列のおかれる領域は文字列型の場合と数値型の場合とでは異なります．静的な文字列配列は文字列記憶領域にとられます．数値型配列は静的・動的を問わず非文字列記憶領域にとられます．文字列記憶領域は，単純型変数などを記憶するDGROUPEと名づけられた領域と同一のセグメントにあります．

注意すべきは，COMMONで宣言された配列はDGROUPE中に領域がとられることです．DGROUPEは最大でも１セグメント(65536バイト)の大きさしかなく，QBのシステム変数もそこを利用するためユーザが使用できるのは，関数FRE(０)で調べると，44Kバイト程度しかないことです．したがって，本書**2.3**の単語帳プログラムでは，600以上とれるはずの配列X( )が350程しかとれません．

1.2 制御構造

従来のBASICに比べて，QBでは制御構文に大幅な機能強化がなされています．WHILE/WENDのほかにDO/LOOP構文が付加され，ループ中からの脱出もEXIT DO，EXIT FOR，EXIT WHILEなどが可能になりました．

分岐は，END IFが許されたため，複数行にまたがって記述できるようになりました．また，SELECT文が新設されて，選択文が可能となりました．

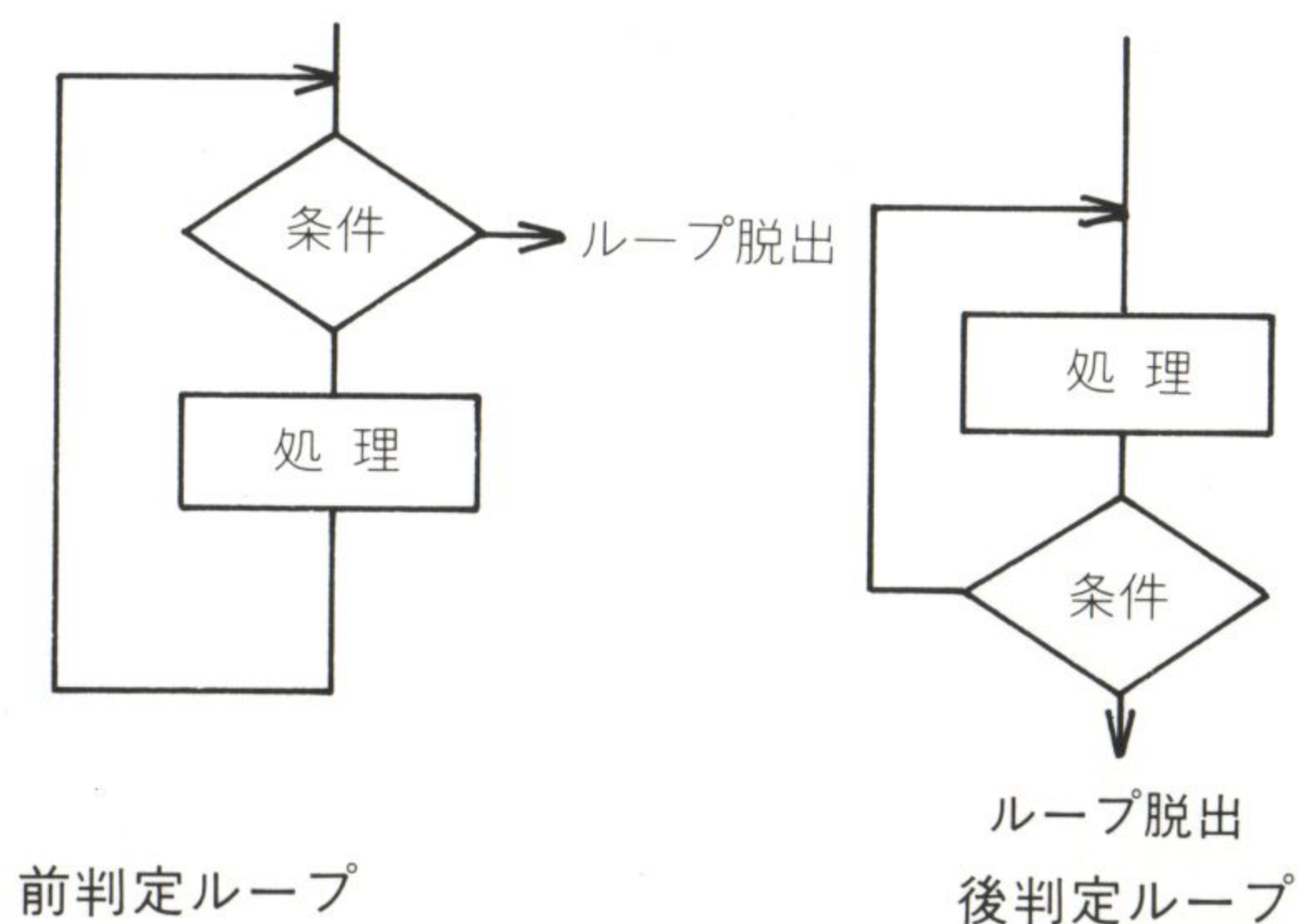
1.2.1 ループ

ループ構造とは，従来のBASICではFOR/NEXTやWHILE/WENDで表された，処理の繰り返し構文のことです．変数の値のみを順に変化させ，処理の手順は変わらないような場合に利用されます．

標準的なループ構文には図1.2.1に示されるようなものがあります．処理の前に処理を行うか否かを判断する前判定ループと，あとで判断する後判定ループがあ

●図1.2.1 前判定ループと後判定ループ

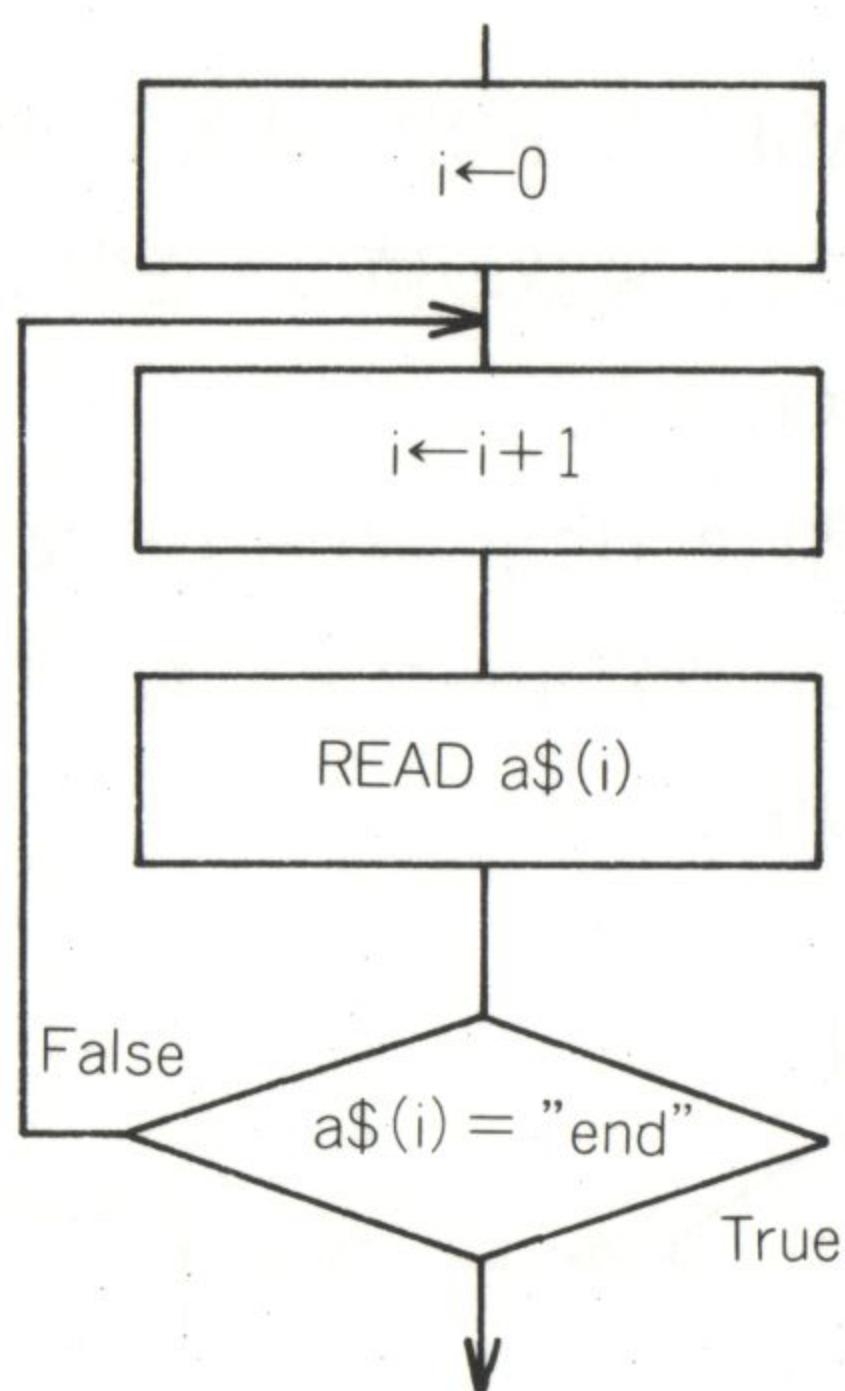
条件
ループ脱出
処 理
前判定ループ

処 理
条件
ループ脱出
後判定ループ

ります．現実には，処理の途中から抜け出たり，脱出条件が複数あったりなどの，いろいろなループが存在します．

はじめに，図1.2.1の型のループについて考えてみます．

BASICで用いられてきたFOR/NEXTやWHILE/WENDループは前判定ループです．したがって，従来のBASICで後判定ループのプログラムを書こうとすると，IF/GOTO文を使うか，WHILE/WENDループが使用できるようにアルゴリズムを前判定型に変更するといったことを行う必要がありました．しかし，QBではDO/LOOP構文のおかげで後判定のプログラムが書きやすくなりました．具体的なプログラム例で説明しましょう．

文字配列a$にREAD/DATAステートメントで，次々と文字データを読み込むプログラムを考えてみます．この繰り返し処理の終了は，文字データの最後に文字列“end”を並べておき，この文字列を読み込んだところでループを抜けるようにします．図1.2.2にフローチャートを示します．リスト1.2.1にはIF/GOTO文を使っ

●図1.2.2 後判定型のループ例

●リスト 1.2.1

```
1: 100 '   list 1.2.1
2: 110 '
3: 120 '
4: 130     I = 0
5: 140 *LOOPTOP
6: 150     I = I + 1
7: 160     READ a$(I)
8: 170     IF a$(I)<> "end" THEN GOTO *LOOPTOP
```

たプログラムを，リスト1.2.2にはWHILE/WEND文を使ったプログラムを示します．

フローチャートからも明らかなように，READ文でデータを読み取ってから終わりの判断を行っていますから，このループは後判定型です．このような後判定型のループを前判定型のWHILE/WEND構文を使ってコーディングするためには，リスト1.2.2で示されるようにループに入る前に，その処理を1回だけ余分に行うようにアルゴリズムを変更しなければなりません．これは，初心者にとってわかりづらいことです．プログラミングの学習でつまずく関所でもあります(FORTRANのみを学んできた人にとっては，かなりのベテランでも難しく感ずるようです)．

それに対して，DO/LOOP構文を用いると，リスト1.2.3からもわかるように，図1.2.2のフローチャートそのままにコーディング可能です．これより，従来のBASICでは後判定ループがコーディングしづらいという点が，QBでは改善されていることがわかります．

このDO/LOOP構文は前判定，後判定のどちらの型のループでも書けます．そのうえ，判定条件にWHILE(～の間繰り返す)でもUNTIL(～まで繰り返す)でも使えるという特徴があります．したがって，DO/LOOP構文では，図1.2.3に示すように4種類のループが可能になります．

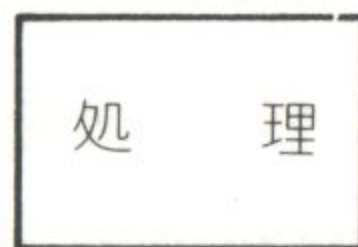
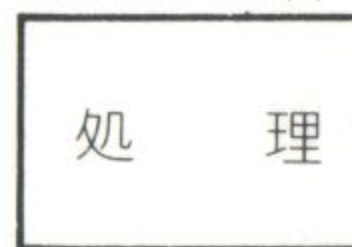
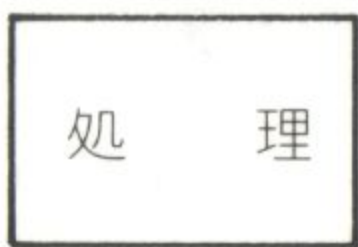
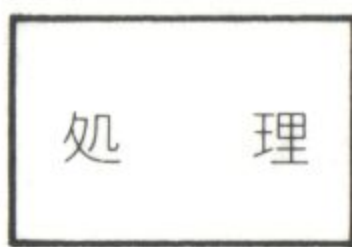
タイプⅠは，従来からの拡張BASICにおけるWHILE/WENDループや，Pascal

●リスト 1.2.2

```
1: '  list 1.2.2
2: '
3: '
4:      i = 1
5:      READ a$(i)
6:      WHILE a$(i) <> "end"
7:          i = i + 1
8:          READ a$(i)
9:      WEND
```

●リスト 1.2.3

```
1: '  list 1.2.3
2: '
3: '
4:      i = 0
5:      DO
6:          i = i + 1
7:          READ a$(i)
8:      LOOP UNTIL a$(i) = "end"
```

●図1.2.3 DO/LOOPの4つのタイプ

のwhileループと，タイプIVは，Pascalのrepeat untilループと同様な働きをします．タイプIIとIIIが，QBで新しく増えたタイプで，これらのタイプを使わないでプログラミングしようと思えばできないことはありません．しかし，使ってみるとなかなか便利なタイプです．

たとえば，タイプIIのDO UNTIL/LOOPについてひとつ例をあげてみます．

シーケンシャルファイルからデータをなくなるまで，つまりファイルのEOFマークを見つけるまで配列に読み込むような処理を行う場合，よくWHILE/WENDループが使われます．

すなわち，

```
100 DIM X$(200)
110 OPEN FILEX$ FOR INPUT AS #1
120 I=0
130 WHILE NOT EOF(1)
140         I=I+1
150         LINE INPUT #1, X$(I)
160 WEND
```

といったようになります.

ここで, WHILE NOT EOF(1)は, ファイル番号1のファイルの“EOF(ファイル終了マーク)を見つけない間は繰り返す” という意味です.

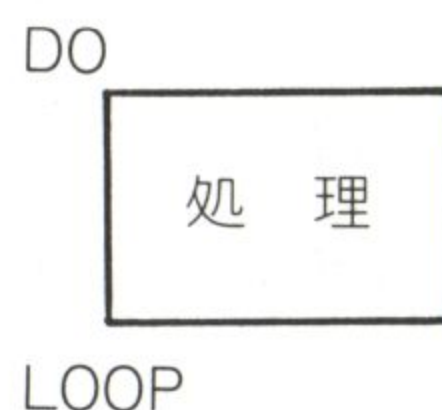
しかし, この意味を変えないように言い換えると, “EOFを見つけるまで繰り返す” となり, こちらのほうがより自然な表現となります. これはタイプIIのDO/LOOP構造で記述できますから, QBでは次のように書けます.

```
DIM X$(200)
OPEN FILEX$ FOR INPUT AS #1
I=0
DO UNTIL EOF(1)
        I=I+1
        LINE INPUT #1, X$(I)
LOOP
```

小さなことかもしれませんが, ループの判定条件に対応して, 表現を変えることなく, より自然にコーディングできることは, プログラムを書く場合も読む場合も重要なことではないでしょうか.

DO/LOOP構造は無限ループを作ることも簡単にできます. 脱出条件をつけずに, 処理の前後をDOとLOOPで挟めば無限ループとなります(図1.2.4).

●図1.2.4 無限ループ

```
DO
┌────────┐
│  処 理  │
└────────┘
LOOP
```

無限ループや, ループ回数があらかじめ定められているFOR/NEXTループなどから, ループを脱出する必要がある場合があります. たとえば, 文字列型配列b$(100)の要素と, ある文字列型変数c$の内容を比較して, 一致したらその文字列型配列の添字をとりだす処理を考えてみます. FOR/NEXTループで配列の最初から最後まで順に配列要素とc$の内容を比較していくとすると, 文字列の内容が一致したあと, 一般的には, 残りの配列要素を調べる必要はないので, その時点でループ

●リスト 1.2.4

```
 1: 100 '    list 1.2.4
 2: 110 '
 3: 120 '
 4: 130 DIM B$(100)
 5: 140 C$ = "ABC"
 6: 150 FOR I = 0 TO 100
 7: 160     IF B$(I)=C$ THEN GOTO *LOOPOUT
 8: 170 NEXT I
 9: 180 *LOOPOUT
10: 190 SUBSCRIPT = I
```

●リスト 1.2.5

```
1: '     list 1.2.5
2: '
3: '
4:      DIM b$(100)
5:      c$ = "ABC"
6:      FOR i = 0 TO 100
7:          IF b$(i) = c$ THEN EXIT FOR
8:      NEXT i
9:      Subscript = i
```

から脱出したくなります．このような場合，従来のBASICでは，IF/GOTO文を使わざるを得ませんでした(リスト1.2.4)．

QBでは，ループの途中から脱出するために，EXIT文が用意されています．EXIT文を使うと，GOTO文を使うことなくループの途中から抜けることができます(リスト1.2.5)．

プログラム例をひとつ，リスト1.2.6にあげておきます．

このプログラムは，20文字までの文字列をキーボードより受けつけ，文字列変数s$に格納するものです．ただし，20文字以内でも，[↵]キーが押されるとそれまで押された文字列がs$に格納され，[ESC](エスケープ)キーが押されると“ESCキーが押されました”という文字列がs$に格納されるというものです．

外側のDO/LOOP UNTILによって，キーボードより入力された文字数が20文字を越えるまで処理が繰り返されます．内側のDO/LOOP UNTILは，キーボードより何かキーを入力させる場合の典型的な方法です．

何かキーが押されるとこの内側のループから抜け，次の2つのIF文によって押されたキーのチェックを行います．

●リスト 1.2.6

```
 1: '    list 1.2.6
 2: '
 3: '
 4: CLS
 5: s$ = ""
 6: n = 0
 7: LOCATE 1, 20
 8: PRINT "なにか英数字キーをおしてください．"
 9:
10: DO
11:     n = n + 1
12:
13:     DO
14:         a$ = INKEY$
15:     LOOP UNTIL a$ <> ""
16:
17:     IF a$ = CHR$(13) THEN EXIT DO
18:
19:     IF a$ = CHR$(&H1B) THEN
20:         s$ = "ＥＳＣキーがおされました．"
21:         EXIT DO
22:     END IF
23:
24:     s$ = s$ + a$
25:
26:     LOCATE 2, 20
27:     PRINT "n ="; n
28:     LOCATE 3, 20
29:     PRINT "a$="; a$
30:     LOCATE 4, 20
31:     PRINT "s$="; s$
32:
33: LOOP UNTIL n >= 20
34:
35: LOCATE 10, 20
36: PRINT "s$="; s$
```

```
IF a$ = CHR$(13) THEN EXIT DO
IF a$ = CHR$(&H1B) THEN
    s$ = "ESCキーが押されました."
    EXIT DO
END IF
```

最初のIF文は，[↵]キーが押されたかどうかの判定をしています．もしそうなら，DOループをEXIT文で抜けます．

2番目のIF文は[ESC]キーが押されたかどうかの判定をしています．その場合は文字列変数s$に“ESCキーが押されました．”というメッセージを代入してDOループを抜けます．

どちらのキーでもない場合は，次の代入文で文字列を連結していきます．

```
s$ = s$ + a$
```

外側のループを抜けるには，２つのキー（↵とESC）が押されてEXIT文が働く場合と，入力文字が20文字を越えた場合に，次のLOOP UNTIL文による場合とがあります．

```
LOOP UNTIL n >= 20
```

20文字入力する間に，２つのキーが押されなかった場合は，この判定が有効になります．

プログラムの実行途中のa$，s$およびnの値と，実行結果としてのs$の値はディスプレイに表示されますので，キーボードからいろいろな値を入力して試してください．

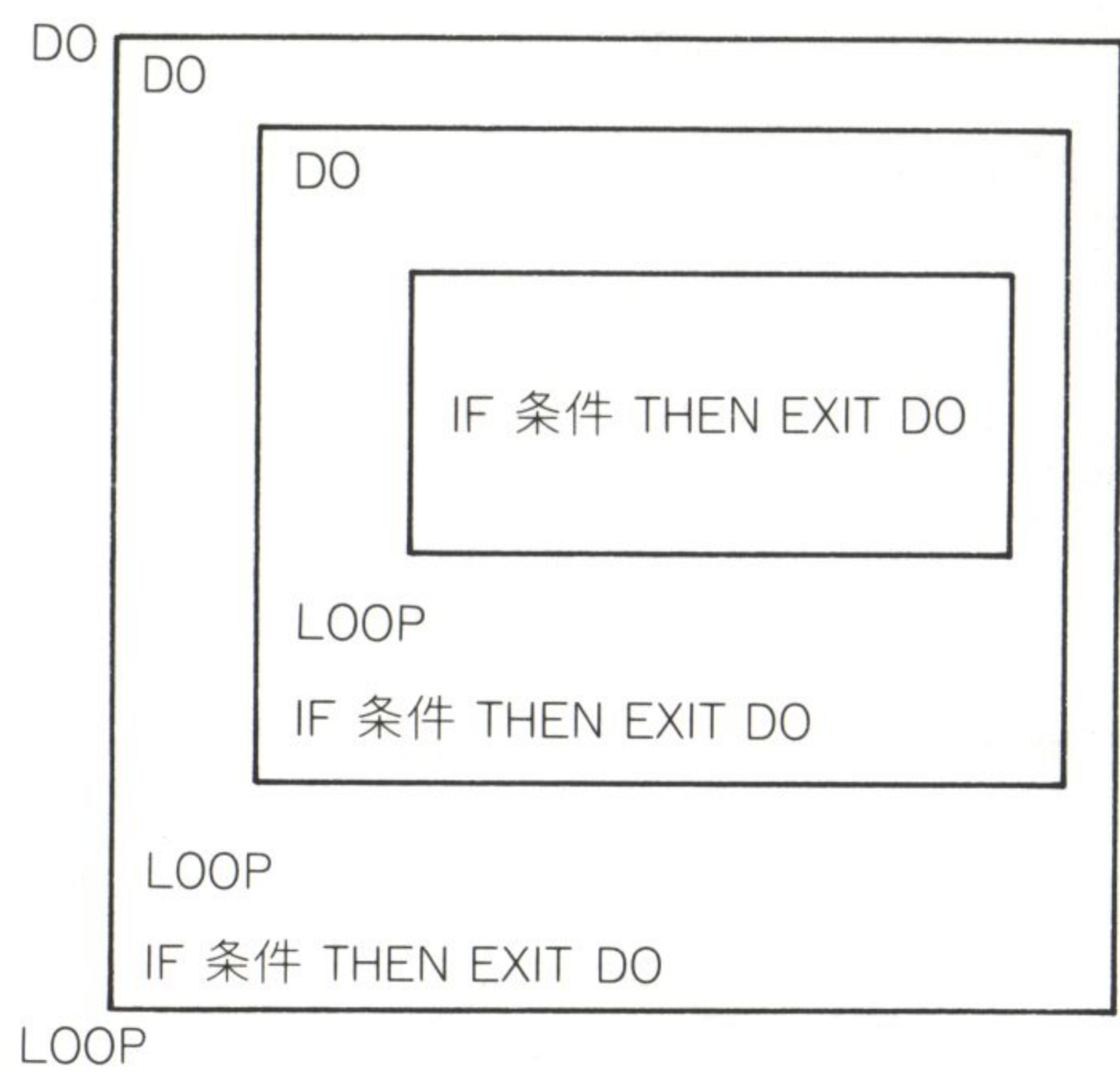
この例からもわかるように，EXIT文を使うとループの途中から条件によって効果的に脱出することができます．

しかし，EXIT文にも機能的にやや弱い点があります．それはループを脱出するときに，EXIT文の存在するループだけからしか脱出できない点です．ループのネストが２重，３重と深い場合，その一番深いループから，あるひとつの条件で一番外側へ脱出するには，図1.2.5に示したように同じEXIT文を３回書かなければなりません．あるいは，ひとつの命令で脱出ということであれば，GOTO文を使わざるをえません．ひとつのEXIT文ですべてのループを脱出できれば，たいへん便利だし，プログラムの可読性も向上するでしょう．

以上に述べたように，ループ構造に対してのDO/LOOPと，それからの脱出のEXIT文がQB上で強化された点です．従来のMBASICとの互換性からWHILE/WENDも残されていますが，DO WHILE/LOOPに置き換えられますので，プログラミングスタイルからはDO/LOOPに統一したほうがよいと思います．ちなみに本書では，そうしています．

もうひとつのFOR/NEXTループは，ループ変数が重要な役割を果たすループ，たとえば配列の内容を順に処理するような場合には有用です．したがって，この２種類のループが使用できれば充分でしょう．

●図1.2.5 多重DOループからEXIT文による脱出

```
DO
  DO
    DO
      IF 条件 THEN EXIT DO
    LOOP
    IF 条件 THEN EXIT DO
  LOOP
  IF 条件 THEN EXIT DO
LOOP
```

1.2.2 分　岐

分岐構造とは，BASICではIF/THEN/ELSEを使って表される処理の枝別れ構文のことです．条件によってある処理を行わせたり，処理の流れを2方向に分岐させたい場合に利用されます．

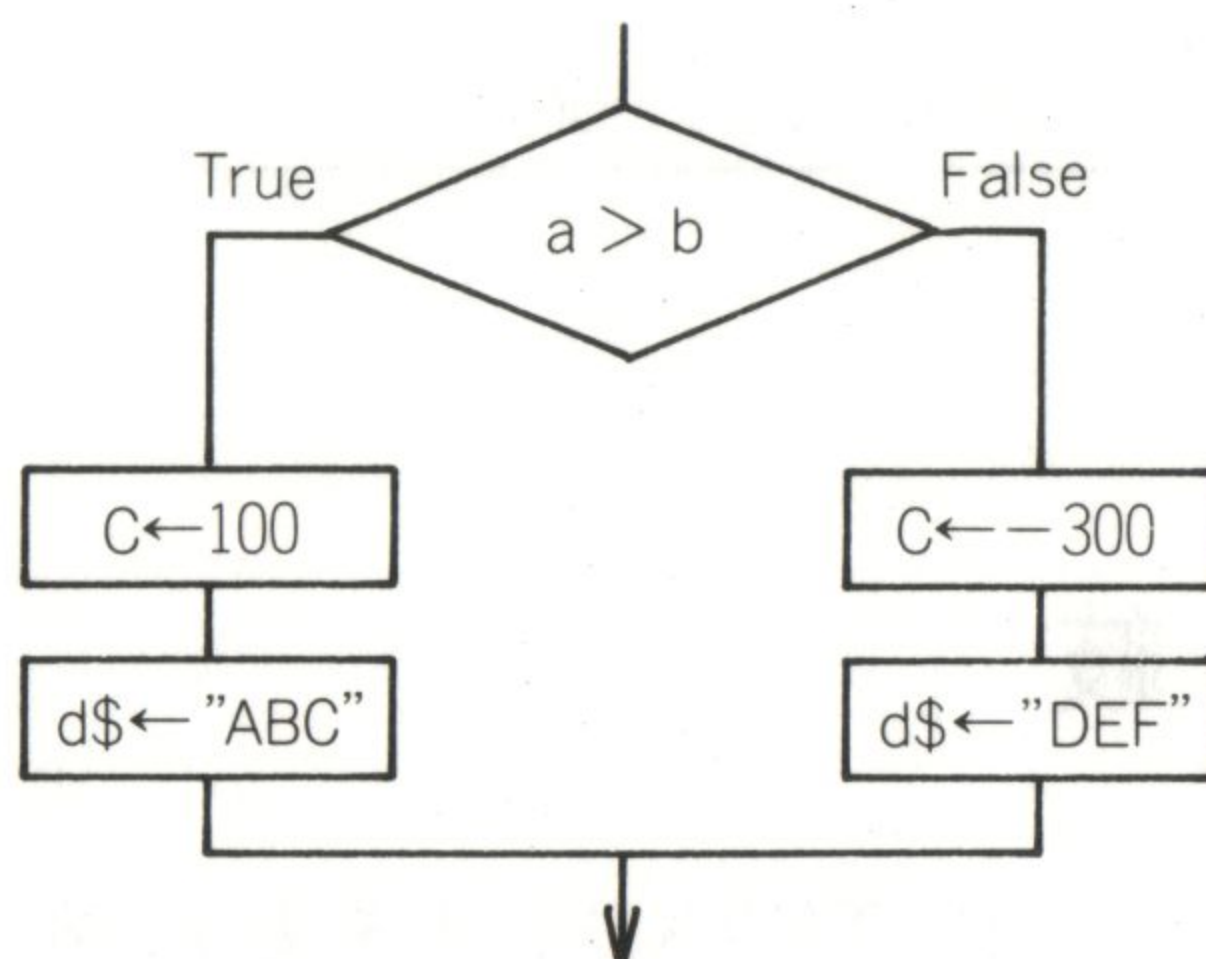
従来のBASICでは，この分岐構造の記述能力がPascalやCに比べて使いにくいものでした．なぜならば，処理の流れを2つに分岐したあとの処理のブロック化が困難であったからです．すなわち，IF/THEN/ELSE文を1行に書かねばならなかったために，分岐後の処理が複数にわたるときはGOTO文やGOSUB文に頼るほかはありませんでした．リスト1.2.7のプログラムは，単純な分岐構造(図1.2.6)を従来のBASICでコーディングしたものですが，GOTO文を2か所も使用せざるをえませんでした．

あるいは，同じ内容のものをIF/THEN/ELSEを1行で用いて，リスト1.2.8のように2個のGOSUBを用いて書くことになります．この場合には，サブルーチン＊L3と＊L4を読まないと処理の内容がわからず読みやすくありません．

しかし，QBでは，END IF文が使え，複数行にわたって記述できるようになったため，この問題は解決しました．図1.2.6をQBでコーディングするとリスト1.2.9

●リスト 1.2.7

```
 1: 100 '    list 1.2.7
 2: 110 '
 3: 120 '
 4: 130 IF A>B THEN GOTO *L1
 5: 140     c = -300
 6: 150     d$ = "DEF"
 7: 160     GOTO *L2
 8: 170 *L1
 9: 180     c = 100
10: 190     d$ = "ABC"
11: 200 *L2
```

●図1.2.6 簡単な2分岐構造

●リスト 1.2.8

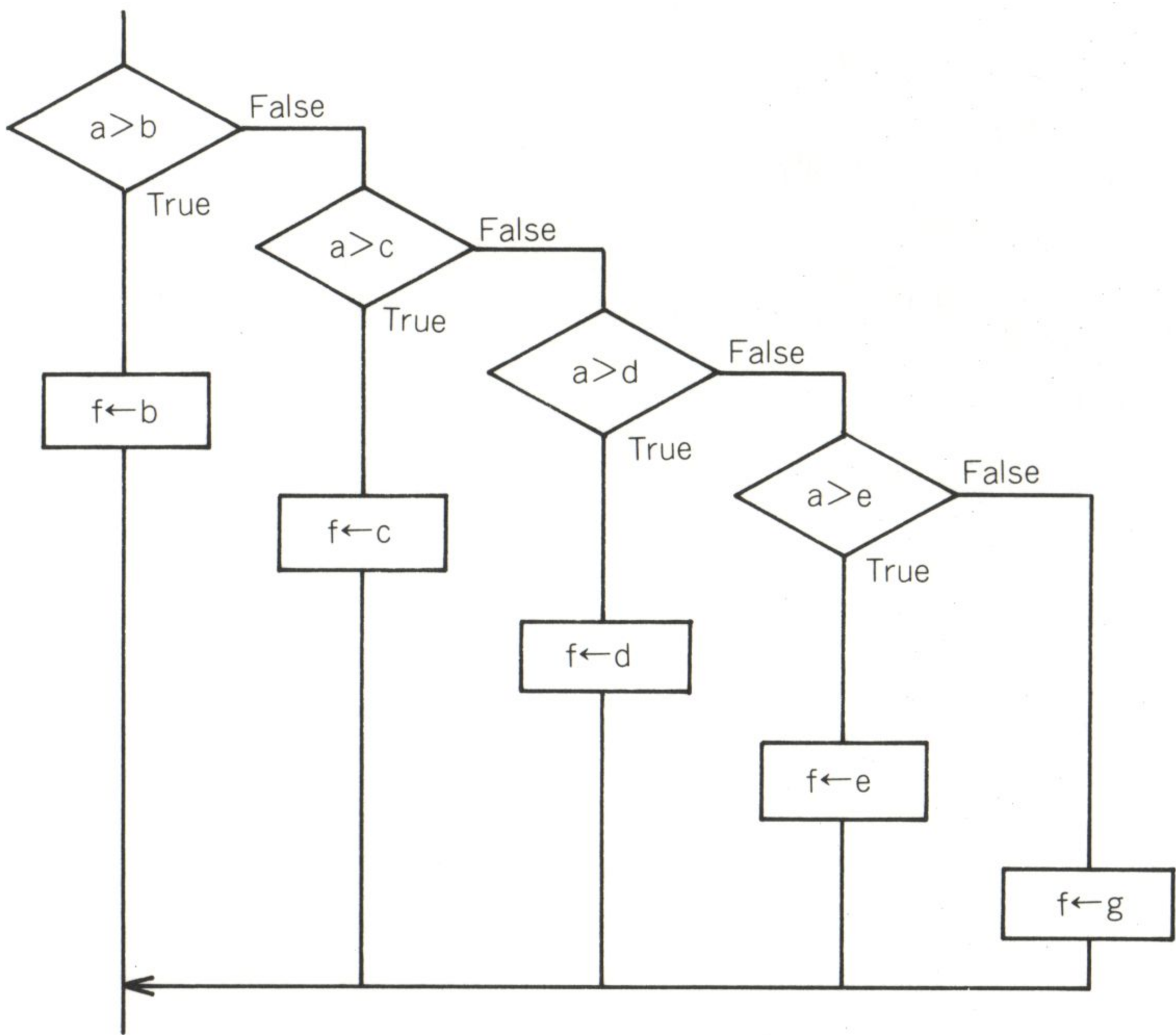
```
 1: 100 '    list 1.2.8
 2: 110 '
 3: 120 '
 4: 130 IF A>B THEN GOSUB *L3 ELSE GOSUB *L4
 5:
 6:
 7: 500 *L4
 8: 510     c = -300
 9: 520     d$ = "DEF"
10: 530 RETURN
11:
12:
13: 700 *L3
14: 710     c = 100
15: 720     d$ = "ABC"
16: 730 RETURN
```

のようになり，リスト1.2.7やリスト1.2.8に比べプログラムが見やすくなりました．

このQBのブロックIF文は，PascalやCのIF文と同様にIFの入れ子構造が可能です．図1.2.7に示すフローチャートをブロックIF文の入れ子構造を使ってコーディングすると，リスト1.2.10のようになります．しかし，入れ子構造があまり深いとプログラムの可読性が悪くなってきます．

●リスト 1.2.9

```
 1: '     list 1.2.9
 2: '
 3: '
 4:     IF a > b THEN
 5:         c = 100
 6:         d$ = "ABC"
 7:     ELSE
 8:         c = -300
 9:         d$ = "DEF"
10:     END IF
```

●図1.2.7 多分岐構造

QBでは，ブロックIF文のオプションとしてELSEIF文がありますので，このような場合にも対処できます．同じフローチャートのものをELSEIF文を使用してコーディングすると，リスト1.2.11のようになり，入れ子構造なしでもプログラミングができます．

なお，ブロックIF文の登場で1行のIF/THEN/ELSE文はほとんど使わなくなると思いますが，分岐後の処理が1命令でかつ短い場合は，あえてブロックIF文にするより見やすいと思われますので，第2部ではそのような場合にかぎり，1行のIF文を使っています．

●リスト 1.2.10

```
 1: '     list 1.2.10
 2: '
 3: '
 4: a = 10
 5:
 6: b = 70: c = 100: d = 50: e = 40
 7:
 8:
 9: IF a > b THEN
10:    f = b
11: ELSE
12:    IF a > c THEN
13:        f = c
14:    ELSE
15:        IF a > d THEN
16:            f = d
17:        ELSE
18:            IF a > e THEN
19:                f = e
20:            ELSE
21:                f = g
22:            END IF
23:        END IF
24:    END IF
25: END IF
26:
27: PRINT f
```

●リスト 1.2.11

```
 1: '     list 1.2.11
 2: '
 3: '
 4:  a = 65
 5:
 6:  b = 70: c = 60: d = 50: e = 40
 7:
 8:
 9:  IF a > b THEN
10:     f = b
11:  ELSEIF a > c THEN
12:     f = c
13:  ELSEIF a > d THEN
14:     f = d
15:  ELSEIF a > e THEN
16:     f = e
17:  ELSE
18:     f = g
19:  END IF
20:  PRINT f
21:
```

1.2.3 選　択

選択構文とは，枝別れする条件が多数存在し，それによって処理の流れを多分岐させせようとする構文です．従来のBASICではON GOTOあるいはON GOSUB文を用いて表現されていました．この文を多用すると，簡単な多分岐構造のプログラムでも，GOTO文が錯綜して可読性の悪いものになりがちでした．

QBでは新しく，SELECT CASE/END SELECT構文が加わったため，この問題が改善されています．

ON GOSUBによる選択構文とSELECTによる選択構文の比較をしてみます．リスト1.2.12は前者，リスト1.2.13は後者です．

このプログラムは，キーボードから入力された1文字WHICH$が“1”～“4”ならば，それぞれファイル処理，測定，解析，レポート作成をするサブプログラムを呼び出し実行します．“0”であれば，何もしないで終了します．

ON GOSUBでは，ONの次の式の値は数値1, 2, 3…をとらなくてはならないため，変数WHICH$の値を数値に変換してSELに代入しています．その分，可読性が損なわれています．また，サブプログラム＊FILE，＊MEASなどがタイプミ

●リスト 1.2.12

```
1: 100 '     list 1.2.12
2: 110 '
3: 120 '
4: 130 *AGAIN
5: 140      INPUT WHICH$
6: 150      SEL = ASC(WHICH$) - 48
7: 160      ON SEL GOSUB *FILE, *MEAS, *ANAL, *REPORT
8: 170      IF WHICH$ <> "0" THEN GOTO *AGAIN
9:
```

●リスト 1.2.13

```
 1: '     list 1.2.13
 2: '
 3: '
 4: DO
 5:     INPUT which$
 6:     SELECT CASE which$
 7:         CASE "1"
 8:             call File(     )
 9:         CASE "2"
10:             call Meas(     )
11:         CASE "3"
12:             call Anal(     )
13:         CASE "4"
14:             call Report(     )
15:     END SELECT
16: LOOP UNTIL which$ = "0"
17:
18:
```

スでRETURN文が脱落していたり，コーディングミスで，RETURN文がスキップされていても見出すことが困難です．

QBのSELECT文では，CASEのあとに任意の式が使えるのでwhich$と，そのとるべき値“1”，“2”，などを直接使用でき，見やすくなります．また，この例では，プロシージャFile，Measなどは“モジュール”になっているので，上に述べたようなミスは生じません．

その点からも，QBではプログラムが読みやすく，かつ書きやすくなります．

また，このような選択構文を表すものに，Pascalではcase文が，Cではswitch文があります．3つとも書式のスタイルはよく似ていますが，QBとほかの2つの言語では次の点が大きく異なります．

つまり，QBではCASE節のうしろ(つまり選択子)に式が記述できることに対して，PascalやCでは選択子に定数しか記述することができません．このことは，QBで書かれた次のようなプログラムをPascalやCに書き換える場合に問題になります．

```
CONST right = 1, left = -1, balanced = 0
SELECT CASE TargetWeight
      CASE balanced
            処理ブロック1
      CASE -DirNow
            処理ブロック2
      CASE DirNow
            処理ブロック3
END SELECT
```

ここで，DirNowは値としてleftまたはrightの値をとる変数です．PascalやCでは，QBのように選択子にDirNowのような変数をとることができないため，選択構造が使えず，結局次のようにif文を使わざるを得ないことになります．

```
if TargetWeight = balanced then
  begin
        処理ブロック1
  end
else
  if TargetWeight = -DirNow then
     begin
        処理ブロック2
     end
  else
     if TargetWeight = DirNow then
        begin
         処理ブロック3
        end;
```

このように，選択文の表現に関してはQBに柔軟性がみられます．

これらの例でよくわかるように，QBではSELECT CASE文が加わったため，ON GOTO(あるいはON GOSUB)は実際上必要がなくなりました．

1.3 データ構造

QBでは，従来の文字列，整数，単精度実数，倍精度実数型のほかに，長整数が付加されました．変数の型宣言についてはDEFINTなどのようにまとめて指定する以外に，… AS INTEGERのように，個々の変数ごとに宣言することが可能となりました．

また，記号定数宣言が可能となったので，誤って定数を書き換えるような心配はなくなりました．

最も大きな追加事項は，TYPEによって型宣言ができるようになったことです．PascalやCと比べると，要素に配列が使えないなどの制約がありますが，TYPEのネストが可能であるなど，使いやすくなっています．プロシージャの引数としても使用可能なので，プログラミングの幅が広がります．

1.3.1 記号定数

QBでは，新しく記号定数の宣言ができるようになりました．記号定数に置き換えることができるのは，数値と文字列および変数を含まない式です．キーワードCONSTのあとに，変数の代入文のような要領で次のように書きます．

```
CONST InfSize = 20
CONST False = 0, True = NOT False
CONST LastData ="END"
```

記号定数名の最後に型宣言文字(%，&，!，#，$)をつけて，明示的に記号定数の型を宣言することも可能です．つけない場合は，記号定数を代入する変数に合うように，最適な型が自動的に決められます．上の例では，InfSize，False，Trueの3つの記号定数は整数型，LastDataは文字列型になります．

記号定数名のつけ方は変数名のつけ方に準じますが，有効範囲は次のようになります．

同一モジュール内の記号定数の有効範囲は，その宣言された場所がモジュールレベルコードか，SUBあるいはFUNCTIONプロシージャ内かによって異なります．

モジュールレベルコードにおいて，宣言された記号定数の有効範囲はグローバルで，そのモジュールレベルコード内はもちろん，モジュール内のすべてのSUBあるいはFUNCTIONプロシージャ内で有効です．したがって，モジュールレベルコードで宣言された記号定数名は，モジュール内のプロシージャで同じ記号定数名を別の目的で使用することはできません．

SUBあるいはFUNCTIONプロシージャ内で宣言された記号定数は，ローカルで，そのSUBあるいはFUNCTIONプロシージャ内だけで有効となります．

複数のモジュールからなる場合，記号定数の情報は渡されません．すなわち，メインモジュールのモジュールレベルコードで宣言された記号定数は，メインモジュール内ではグローバルですが，サブモジュールにその情報が渡らないため，サブモジュールのモジュールレベルコードでも宣言しておく必要があります．

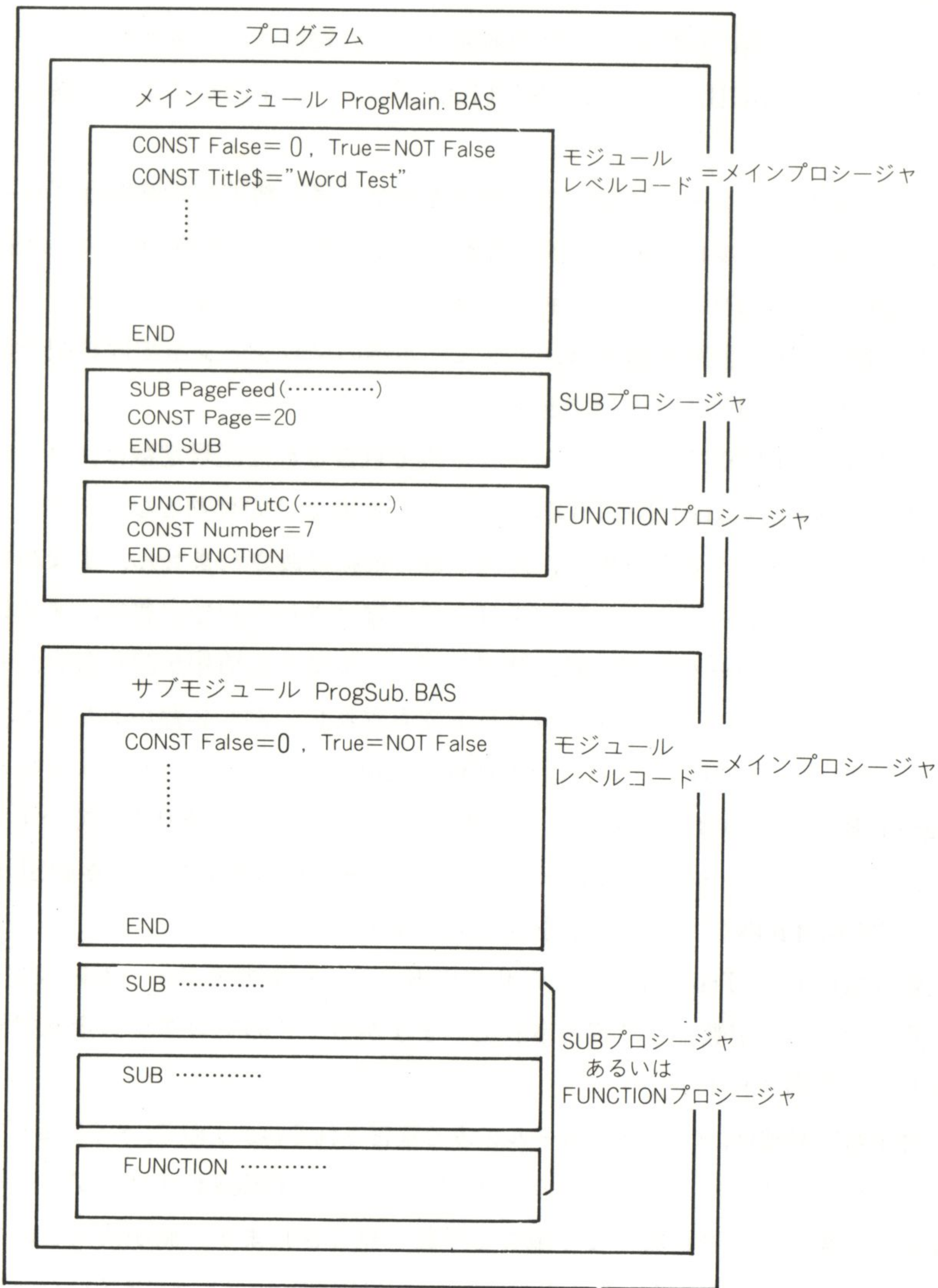
図1.3.1に示したプログラムは，ProgMain.BASというメインモジュールと，ProgSub.BASというサブモジュールの 2 つのモジュールから成り立っています．

それぞれのモジュールは，モジュールレベルコードのほかにいくつかのSUBあるいはFUNCTIONプロシージャを含んでいます．

記号定数FalseとTrueは，メインとサブの両モジュールのモジュールレベルコードで同一の内容の定数として宣言されていますから，このプログラム全域で同一の値として参照されます．

記号定数Title$は，メインモジュールのみで宣言されていますので，メインモジュール内においてのみグローバルとなります．したがって，Title$をサブモジュール内で使用した場合，記号定数でなく単なる変数と見なされます．値が代入されていないと，Title$の内容は空のままです．また，記号定数Pageは，SUBプロシージャPageFeed内で，NumberはFUNCTIONプロシージャPutC内でそれぞれ宣言されていますから，その各プロシージャ内でのみ有効となります．

記号定数を上手に利用すると，プログラムは読みやすくすっきりしますから，QBでは積極的に利用するとよいでしょう．

●図1.3.1 2つのモジュールからなるプログラム

1.3.2 型宣言

BASICで使用できる変数の型は，大別すると数値変数と文字列変数に分けられます．QBでは数値変数に新たに長整数型が加わり，文字列変数には固定長文字列型が追加されました．

型の宣言方法は，従来のBASICと同様に，変数名の最後に型を示す５種類の型宣言子(%，&，!，#，$)をつけるのが一般的です．型宣言子を省略した場合は，単精度実数型(すなわち！をつけたものと同じ)になります．また，$をつけて文字列型を宣言した場合は可変長文字列になり，固定長文字列型を宣言するにはあとで述べるDIM文を使います．

また，プログラムの先頭で使うことによって，変数名の先頭の文字で変数の型を定義する事ができる命令に，DEFINT(整数)，DEFLNG(長整数)，DEFSTR(文字列)，DEFSNG(単精度実数)，DEFDBL(倍精度実数)の５種類があります．

たとえば，

```
DEFINT I-N
```

と宣言すれば，変数名の先頭がIからNまでで始まるものはすべて型宣言子%をつけなくても整数型になります．

ただし，型宣言子をつけた変数はこの定義より優先しますから，たとえば，

```
ITEM$="ABC"
```

のITEM$は変数名の先頭はIですが，文字列型変数となります．

さて，QBでは新たにDIM文を使って変数の型宣言ができるようになりました．書き方は，

```
DIM　変数名　AS　型名
```

です．型名として許されているのは，INTEGER(整数)，LONG(長整数)，SINGLE(単精度実数)，DOUBLE(倍精度実数)，STRING(文字列)，そして次の節で説明するユーザ定義型です．

例を使って説明してみましょう．

```
DIM Month AS INTEGER
DIM PI AS DOUBLE
DIM OneLine AS STRING
DIM OneWord AS STRING * 20
```

この４つの宣言文でMonthは整数型，PIは倍精度実数型，OneLineは文字型のそれぞれ変数として定義されています．OneWordは，固定長文字列として宣言する場合の例で，この場合は20文字の長さとして宣言されています．文字数の最大値は可変長文字列の場合の最大値32,767文字です．

固定長文字列変数に，その文字数より多い文字数を代入すると，多い文字は切り捨てられます．また短い文字列の場合は左から詰められ，あまった文字は空白で埋められます．

この方法の宣言で注意する点は，変数名に型宣言子をつけるとエラーになることです．型宣言子をつけた変数とDIMで宣言した変数を混在して使う場合は注意が必要です．

DIM文はこのように変数の型宣言にも用いますが，従来のBASICでは配列変数の宣言に用いられてきました．QBでも，もちろん配列の宣言にも使います．QBでは配列の添字の範囲を指定できることが新しい点です．

たとえば，

```
DIM WeekDay(1 TO 7) AS STRING
DIM Temp(-10 TO 40) AS SINGLE
```

といったぐあいに，配列変数の型と添字の範囲が指定できます．

もちろん，次のように従来どおりの配列宣言も可能です．

```
DIM Subject$(20)
DIM Number(100)
```

この場合の配列の大きさは，Subject$(0)～Subject$(20)とNumber(0)～Number(100)となります．なお，OPTION BASE文はQBでも有効ですので，配列の添字の最小値を１にすることもできます．

また, DIM文ではSHAREDオプションをつけることによって, 変数をグローバル変数として宣言することができます.

たとえば,

```
DIM SHARED Title$
DIM SHARED Number(100) AS INTEGER
```

のように宣言された場合, 可変長文字列変数Title$と, 整数型配列Numberは, ひとつのモジュール内でグローバルとして扱われます. したがってこれらの変数や配列は, 宣言されたモジュール内のすべてのSUBプロシージャやFUNCTIONプロシージャにおいて, 引数の受渡しなしで, 値の参照や書き換えが可能となります. なお, この宣言はモジュールレベルコードにおいてなされなければなりません.

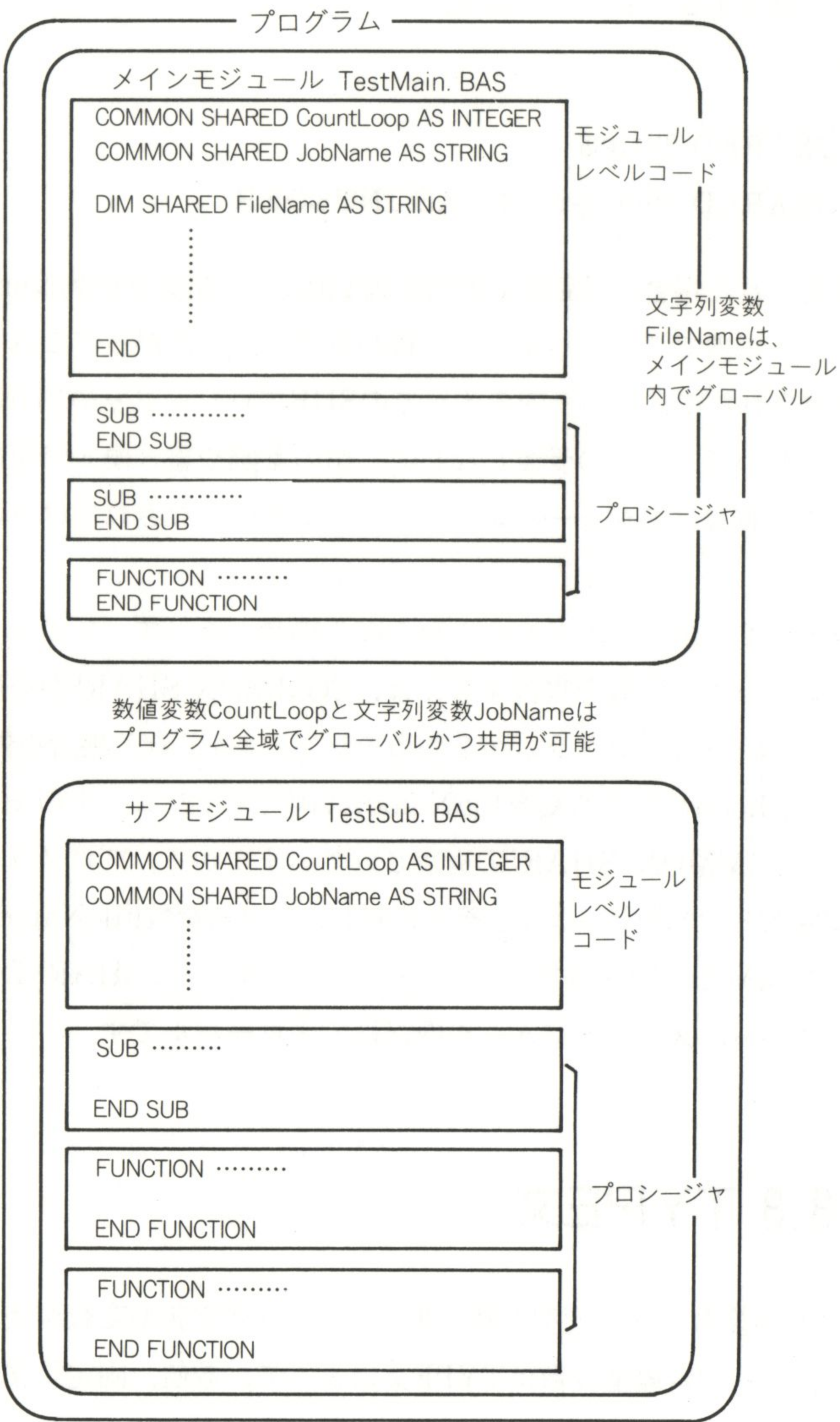
プログラムが2つ以上のモジュールからなる場合, そのすべてのモジュールにわたってグローバルな変数を宣言するには, COMMON SHAREDを使います. 図1.3.2に示すように, プログラムが2つのモジュールからなる場合を例にとると, 数値変数CountLoopと文字列変数JobNameは, 両方のモジュールのモジュールレベルコードでCOMMON SHARED宣言されていますから, プログラム全域でグローバルな変数として扱われます. それに対し, 文字列変数FileNameは, メインモジュールTestMain.BASの中のモジュールレベルコードでSHARED宣言されているのみですから, メインモジュール内だけでグローバルです.

1.3.3 TYPE文

QBでは, 数値変数と文字列型変数に加えて, ユーザ定義型変数が使えるようになりました. ユーザ定義型変数はTYPE文によって, 数値, 固定長文字列, またはユーザ定義型の複合したデータからなる, 新たな型を宣言することにより定義される型を持つ変数で, レコード変数とも呼ばれます.

このレコード変数を使うには, まずTYPE文で型の宣言をする必要があります.

たとえば,

●図1.3.2 COMMON SHAREDとDIM SHAREDの違い

```
TYPE node
    inf AS STRING * 20
    left AS INTEGER
    right AS INTEGER
    weight AS INTEGER
END TYPE
```

のように宣言すると，このユーザ定義型nodeは，20文字の固定長文字列の要素inf，整数型の要素left，right，およびweightの 4 つから成り立っていることが定義されたことになります．要素の型の宣言で文字列型は固定長文字列型に限られることに注意してください．

そして，次にDIM文を使って，レコード変数の宣言をします．

```
DIM Word AS node
DIM X(100) AS node
```

これによって，変数Wordと配列Xは，それぞれnode型のレコード変数として宣言されたことになります．レコード変数の各要素の参照，あるいは要素への値の代入は次のようにします．

```
a$ = Word.inf
b% = Word.left
c$ = x(j).inf

Word.inf = "intellect"
FOR i = 0 to 100
      x(i).weight = 0
NEXT i
```

QBにおけるユーザ定義型のさらに便利な点は，型のネスティングができることです．

たとえば，

```
TYPE  DayTime
    month AS INTEGER
    day AS INTEGER
END TYPE

TYPE  Today
    totalday AS INTEGER
    schedule AS STRING * 30
    calendar AS DayTime
END TYPE
```

のように,最初のTYPE文でDayTime型を宣言しておけば,次のTYPE文で,Today型の要素calendarの型をDayTime型として定義できることになります.

次に示すようにDIM文でToday型のレコード変数を宣言すれば,

```
DIM ChristmasDay AS Today

ChristmasDay.calendar.month = 12
ChristmasDay.calendar.day = 25
```

といったような使い方もできます.

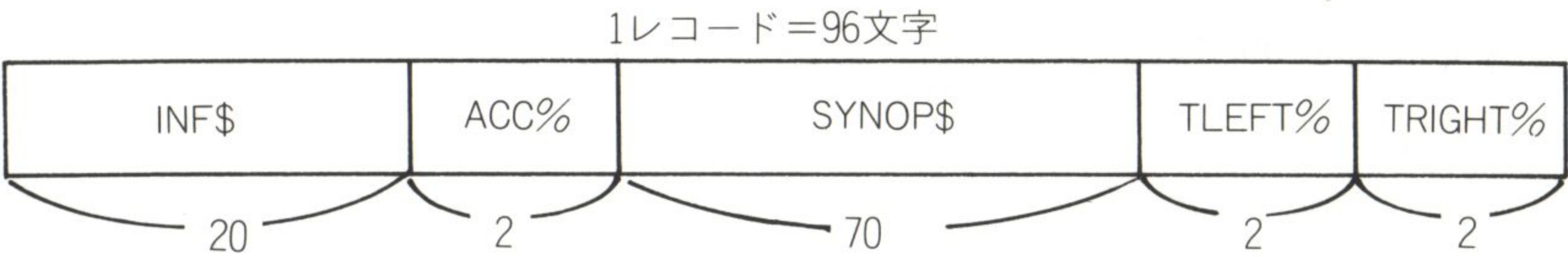
ただし，TYPE文の要素に配列を用いることはできません．このことは改良してもらいたい点のひとつです.

さて，BASICでレコード変数が使えるようになると，便利に感じられることにランダムアクセスファイルがあります.

ランダムアクセスファイルの扱いは,シーケンシャルファイルの場合と異なり,ファイルの1レコードは固定長で定義する必要があります．また，その1レコードの中の各フィールドも固定長で定義しなければなりません.

たとえば,図1.3.3に示すようなレコードを考えます．このレコードをアクセスするランダムファイルを使用するときは，まず1レコードの中の各フィールドを定義しなければなりません.

その場合,レコード変数が使えない従来のBASICではFIELD文を使います.FIELD

●図1.3.3 1レコードの構成図

文を使う場合，変数ACC，TLEFT，TRIGHTなどの数値項目は，すべて文字列項目として定義しなければなりません．

次にファイルのオープンとレコード長の指定をする必要があります．1レコードの長さはプログラマが自分で計算しなければなりません．この場合は96になります．

さらに，ランダムアクセスファイルへのアクセスは，PUTとGETで行うわけですが，その際入出力に応じてデータを，フィールドの左詰め，あるいは右詰めにしてから行わなければなりません．数値データと文字列データの変換操作も必要となります．ここまでの一連の手順をリスト1.3.1に示します．

●リスト 1.3.1

```
 1: '     list 1.3.1
 2: '
 3: '
 4: '      フィールドの定義
 5: FIELD #1, 20 AS INF$, 2 AS ACC$, 70 AS SYNOP$, 2 AS TLEFT$, 2 AS TRIGHT$
 6:
 7: '       ファイルのオープン
 8: OPEN "RNDFILE.DAT" FOR RANDOM AS #1 LEN = 96
 9:
10: '
11: INFVAL$ = "intellect"
12: ACCVAL = 1
13: SYNOPVAL$ = "知性"
14: ・・・・・
15:
16: '      レコードの格納
17: LSET INF$ = INFVAL$
18: LSET ACC$ = MKI$(ACCVAL)
19: LSET SYNOP$ = SYNOPVAL$
20: LSET TLEFT$ = MKI$(TLEFTVAL)
21: LSET TRIGHT$ = MKI$(TRIGHTVAL)
22:
23: PUT #1
24:
25:
26:
27:
```

一方，同じ作業をTYPE文を使って行った場合の例をリスト1.3.2に示します．これを使えば，従来初心者には使用が難しいといわれていた，ランダムアクセスファイルが比較的手軽に使えるのではないでしょうか．

●リスト 1.3.2

```
 1: '    list 1.3.2
 2: '
 3: '
 4: '    フィールドの定義
 5:      TYPE node
 6:        inf AS STRING * 20
 7:        acc AS INTEGER
 8:        synop AS STRING * 70
 9:        tleft AS INTEGER
10:        tright AS INTEGER
11:      END TYPE
12:
13: '    フィールドのオープン
14:      DIM OneWord AS node
15:      OPEN "RNDFILE.DAT" FOR RANDOM AS #1 LEN = LEN(OneWord)
16:
17: '    レコードの格納
18:      OneWord.inf = "intellect"
19:      OneWord.acc = 1
20:      OneWord.synop = "知性"
21:      ・・・・・
22:
23:      PUT #1, OneWord
24:
25:
```

1.4 モジュール

QBでは，プログラムのモジュール化が可能になったことが大きな特徴のひとつですが，モジュールそのものの定義はたいへんわかりにくく書かれています．

Ver.4.5のHandbook第9章“プログラムのモジュール化”では，モジュールの定義が書かかれておらず，いきなりモジュール化の利点から始まっています．その章をずっと読んでいくと，やっと，プロシージャを1個以上含むファイルがモジュールであることがわかります．

ただし，実行の始まる“メインプログラム”，QBではモジュールレベルコード，と呼んでいる部分は，プロシージャではありませんが，それを含むファイルをメインモジュールと呼んでいます．

プロシージャを使用する場合に問題になるのは，宣言やイベントトラップなどがどこまでおよぶかという問題です．以下に，それらについてまとめてみます．

1.4.1 モジュール作成

QBでモジュールと呼ぶプログラム単位は，ディスクにひとつのファイルとしてセーブあるいはロードする単位と考えてよいでしょう．そして，複数のモジュールからなるプログラムの場合，そこから実行が始まるモジュールを“メインモジュール”，そのほかのモジュールを“サブモジュール”あるいは単に“モジュール”と呼びます．複数のモジュールの中でどれがメインモジュールかは，メイク(MAK)ファイルと呼ばれるファイルに書き込まれるわけですが，プログラムをモジュール化して作る場合に，プログラマはどのモジュールをメインモジュールにしたいかを決めておかなければなりません．なぜなら，プログラムはメインモジュールのモジュールレベルコードから実行が始まるので，サブモジュールのモジュールレベルコードに書かれた実行文は実行されません．それによって，モジュール化

をする前には予想しなかったバグを発生させてしまう恐れがあるからです．

プログラムをモジュール化するには，一般的に次の2つの方法があります．

ひとつはプログラムを書いている途中でサブモジュールを作成する方法，もうひとつはそれぞれのモジュールを単独のプログラムとして作っておき，あとで連結する方法です．順に説明しましょう．

前者についてまず述べると，QB環境の中でプログラミングをしているときに，新しくモジュールを作る方法です．まず，QB環境の[ファイル]メニューを開き，その中の[サブファイル作成]機能を選択します．この項目は，メニューがショートメニューですと表示されませんので，その場合は，[オプション]メニューをオープンして，[フルメニュー]オプションをオンにしておいてください．

そうすると，ファイル名とファイルの種類を入力する画面が現れますから，それぞれ入力します．ここでいうファイル名がつまり“サブモジュール名”ということになります．ファイルの種類はディフォルトが“モジュール”になっていますから普通はファイル名だけを入力して[↵]キーを押せばよいことになります．

画面は，そのモジュール作成用のエディタ画面になっていますから，サブモジュールの内容をプログラミングします．

この方法でモジュールを作った場合，新しく作ったモジュールはすべて“サブモジュール”で，もとのモジュールが“メインモジュール”になります．

どのファイルがメインモジュールでどのファイルがサブモジュールかを知りたいときには，ファンクションキー[f·2]を押せばわかります．カーソルを移動させることによりプロシージャの種類が表示されます(図1.4.1)．

なお，プログラムのセーブを行うときは，メニューの中の[すべて保存]という項目を選びます．すべてのプロシージャがセーブされると同時に，プロシージャの連結情報の書かれたメイクファイル(ファイルの拡張子が.MAKのもの)がディスクに書き込まれます．これをMS-DOSのTYPEコマンドで見てみると，プログラム中のモジュール名が順番に書かれていることがわかります(図1.4.2)．この中の一番上に書かれているモジュールがメインモジュールです．

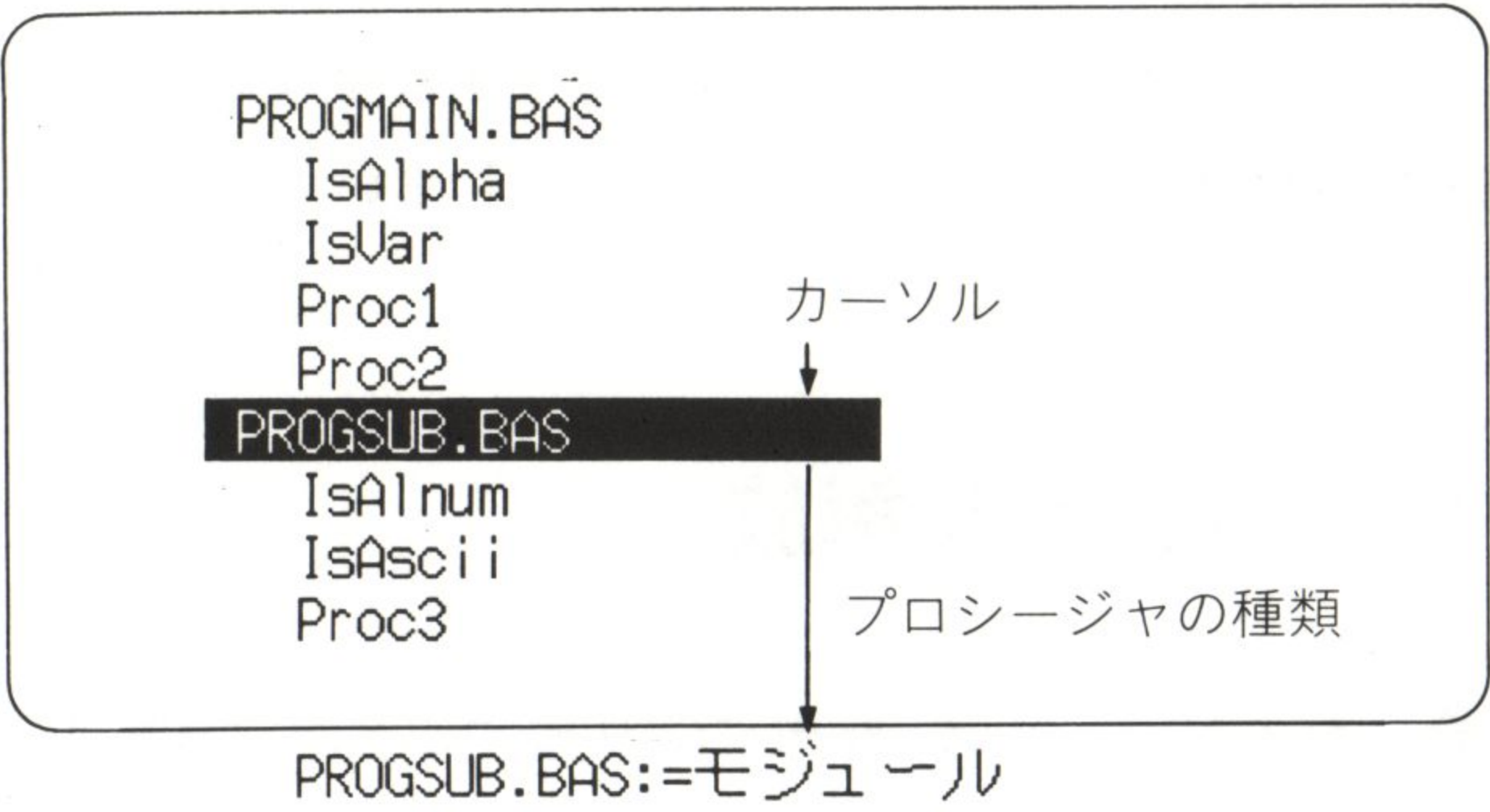
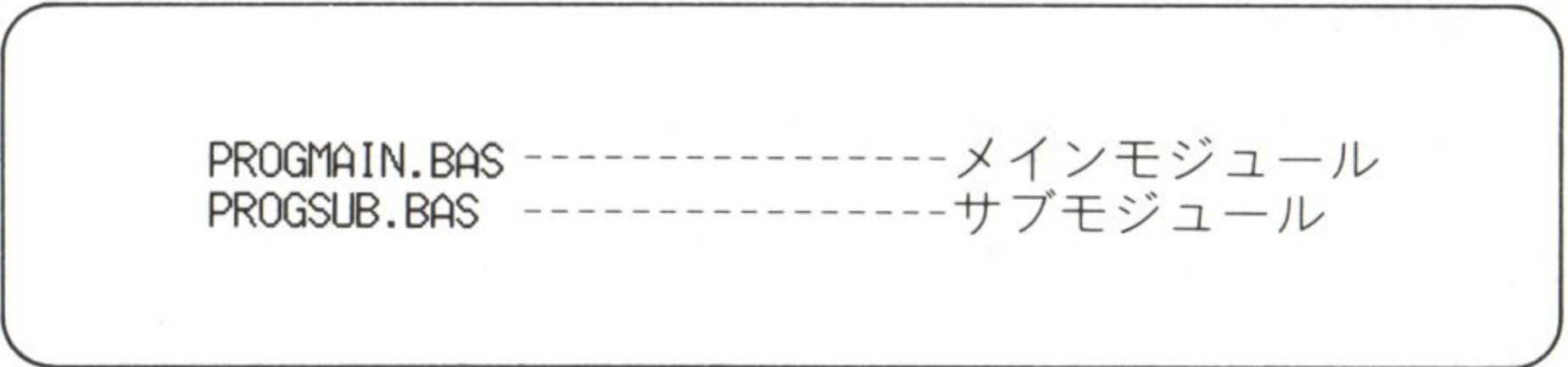
さて，次に後者の方法ですが，複数のプログラムをそれぞれ単独のファイルとしてディスクにセーブしておき，あとで必要なモジュールを連結してプログラム全体を完成する方法です．

モジュールを作る場合，QBのエディタでプログラムを作ったあと，モジュール

●図1.4.1 f･2キーを押した画面

```
PROGMAIN.BAS
  IsAlpha
  IsVar
  Proc1            カーソル
  Proc2               ↓
PROGSUB.BAS
  IsAlnum
  IsAscii
  Proc3            プロシージャの種類

PROGSUB.BAS:=モジュール
```

●図1.4.2 PROGMAIN. MAKをTYPEコマンドで表示

```
PROGMAIN.BAS ----------------メインモジュール
PROGSUB.BAS  ----------------サブモジュール
```

として独立させたい単位で適当な名前をつけてディスクにセーブしておきます．プログラムを構成するモジュールをすべて別のファイル名でセーブし終わったら，メインモジュールにしたいファイルを最初にQBのエディタにロードします．次にQB環境の[ファイル]メニューを開き，その中の[サブファイル読み込み]機能を選択します．[サブファイル読み込み]機能を選択すると，ファイル一覧が表示され，どのファイルをサブモジュールにするかを問い合わせてきますから，カーソルキーでファイル名を選択すれば，サブモジュールとして登録されます．メニューにモジュールという語句が表示されないので，最初は戸惑うと思いますが，サブファイルという語句が，マニュアルでいうところのサブモジュールを指しているようです．サブモジュールが複数あれば，この要領で次々とディスクからファイルを読み込み，サブモジュールとして登録します．

ここで注意しなければいけないことは，読み込むモジュールの中のプロシージャ名と，すでに読み込まれたモジュールの中のプロシージャ名に同名のものがあった場合，読み込むことができないことです．言い換えると，複数のモジュールからなるプログラムの場合も，ひとつのモジュールからなるプログラムと同様に，

プログラム内でプロシージャ名の重複が許されないという点です．したがって，モジュール化を意識したプログラミングを行う場合には，プロシージャ名のつけ方も慎重に行わないと，モジュールを結合するときになって，プロシージャ名の変更といった面倒な問題にぶつかることになってしまいます．

1.4.2 モジュール解放

いったん結合したモジュールは，簡単に切り放すことができます．まず，［ファイル］メニューを開き，その中の［サブファイル解放］機能を選択します．すると，ファイル一覧が表示され，どのモジュールをプログラムから切り放すかを問い合わせてきますから，カーソルキーでファイル名を選択すれば，そのモジュールはプログラムから解放されます．

なお，複数のモジュールからなるプログラムの各モジュール名を変更して別ファイルとして保存したい場合，1回のコマンドですべてのモジュールをセーブすることはできません．ひとつずつモジュールをウィンドウに呼びだしたのち，［ファイル］メニューを開き，その中の［別のファイル名で保存…］機能を選択します．このとき，メインモジュールのモジュール名は最後に変更してセーブしてください．そうしないと，変更したモジュール名の情報がメイクファイルにわたらないため，次回にメインモジュールをロードしたときに，自動的に読み込まれるサブモジュールは，古いモジュール名のものになってしまいます．

もし，そのようになってしまったときは，古いモジュールを［サブファイル解放］機能によって，メインモジュールより切り放し，新しいモジュールを［サブファイル読み込み］機能を使って連結し直してください．

1.4.3 宣言の有効範囲

プログラムを複数のモジュールに分割して構成する場合は，ひとつのモジュールからなるプログラムの場合に比べ，CONSTやDIM文などの宣言がどの範囲まで有効かはっきりさせておかないと，予想外のバグを発生させることになります．以下に重要と思われる点についてまとめます．

◉CONST

有効範囲は単一モジュール内のみです．したがって，複数のモジュールで同じ定数を必要とする場合は，それぞれのモジュールのモジュールレベルコードにおいて，同じ宣言をする必要があります．

◉SHARED

この宣言も単一モジュール内のみで有効です．したがって，リスト1.4.1に示すような場合，グローバル変数Noは，それぞれのモジュールで独立した変数として

●リスト 1.4.1

```
DECLARE SUB SubTest ()
' list 1.4.1
' メインモジュール  PROGMAIN.BAS

DIM SHARED No AS INTEGER

No = 5
PRINT "メインモジュールでのNoの初期値=", No

CALL SubTest

PRINT "メインモジュールでのNoの最終値=", No

END
'   サブモジュール  PROGSUB.BAS
DIM SHARED No AS INTEGER

SUB SubTest

    PRINT "サブモジュールの入口でのNoの値=", No
    No = 7
    PRINT "サブモジュールの出口でのNoの値=", No

END SUB
```

●図1.4.3

```
メインモジュールでのNoの初期値=          5
サブモジュールの入口でのNoの値=          0
サブモジュールの出口でのNoの値=          7
メインモジュールでのNoの最終値=          5
```

扱われます．メインモジュールでNoの値を5にセットして，サブモジュール内のSUBプロシージャSubTestをCALLしたとします．プロシージャSubTest内で変数Noの値を7に書き換えて，メインモジュールに戻してからprintさせてみると，Noの値は5のままです．図1.4.3にリスト1.4.1の実行結果を示します．

このことから，グローバル変数を複数のモジュールで共有したい場合は，SHARED宣言だけではだめで，次に述べるCOMMON SHARED宣言を用いる必要があります．

◉COMMON SHARED

COMMON SHARED宣言は，グローバル変数をモジュール間で共有させたい場合に使用します．したがって，この宣言をした変数は全モジュール間，すなわちプログラム全域でグローバルな変数になります．この宣言は，CONSTなどと同様に，モジュールごとに必要で，変数を共有したいモジュールのモジュールレベルコードで宣言しておかなければなりません．リスト1.4.1と同じテスト内容で，今度は，変数NoをCOMMON SHARED宣言を使ってグローバル変数とした場合のものをリスト1.4.2に示します．また，実行結果を図1.4.4に示します．これより，サブモジュールで変数Noの値が書き変わった結果が，メインモジュールで参照できることがわかります．

◉配列変数のモジュール間での共有

単純変数のモジュール間での共有は，リスト1.4.2のようにCOMMON SHAREDを使えばよいのですが，配列変数の場合は，あらかじめDIM文で配列の型と大きさを宣言しておく必要があります．たとえば，x(20)という整数型の配列変数を各モジュール間で共有しようとすれば，各モジュールのモジュールレベルコードで次のような宣言をしなければなりません．

●リスト 1.4.2

```
DECLARE SUB SubTest ()
' list 1.4.2
' メインモジュール　PROGMAIN.BAS

COMMON SHARED No  AS INTEGER

No = 5
PRINT "メインモジュールでのNoの初期値=", No

CALL SubTest

PRINT "メインモジュールでのNoの最終値=", No

END
'   サブモジュール　PROGSUB.BAS
COMMON SHARED No  AS INTEGER

SUB SubTest

    PRINT "サブモジュールの入口でのNoの値=", No
    No = 7
    PRINT "サブモジュールの出口でのNoの値=", No

END SUB
```

●図1.4.4

```
メインモジュールでのNoの初期値=           5
サブモジュールの入口でのNoの値=           5
サブモジュールの出口でのNoの値=           7
メインモジュールでのNoの最終値=           7
```

```
DIM x(20) AS INTEGER
COMMON SHARED x( ) AS INTEGER
```

2行目のCOMMON SHARED宣言の配列変数の大きさは省略してかまいません．しかし，1行目の配列の大きさ(この場合は20)は各モジュールで宣言する場合，一致しなければいけません．

1.4.4 モジュール間のエラートラッピングおよびイベントトラッピング

QBにおけるエラートラッピングは,ON ERROR GOTOステートメントとエラー処理ルーチンによって行います．エラー処理ルーチンはモジュールレベルコードにおかなければなりません．複数のモジュールにわたって機能するエラートラッピングを実行するためには,ON ERROR GOTOステートメントとエラー処理ルーチンは，両方ともメインモジュールのモジュールレベルコードに置いておく必要があります．リスト1.4.3は複数のモジュール間のエラートラッピングの例です．ON ERROR GOTOステートメントとエラー処理ルーチンErrTrapは，メインモジュールERRMAIN. BASのモジュールレベルコードに置かれています．プログラムの実行は,すぐにサブモジュールのSUBプロシージャSubProcに移り,フロッピィディスクのBドライブにシーケンシャルファイルをオープンしにいきます.ここで,Bドライブにフロッピィディスクが挿入されていないと,エラー番号71のランタイムエラーが発生し，エラートラッピングが起こります．メインモジュール

●リスト 1.4.3

```
DECLARE SUB subproc ()
'   list 1.4.3
'   メインモジュール      ERRMAIN.BAS

ON ERROR GOTO ErrTrap
CALL subproc
END

ErrTrap:
    PRINT "Nummber of Err="; ERR
    PRINT "フロッピィディスクをセットして，"
    PRINT "何かキーを押してください．"
    DO
    LOOP UNTIL (INKEY$ <> "")
RESUME
'   サブモジュール   ERRSUB.BAS

SUB subproc

    OPEN "b:test.txt" FOR OUTPUT AS #1
        PRINT #1, "test"
    CLOSE #1

END SUB
```

のモジュールレベルコードに書かれたエラー処理ルーチンにより，エラー発生の警告とエラー番号が表示されます．Bドライブにフロッピィディスクをセットし何かのキーを押すと，RESUMEステートメントの働きによって，エラーを発生した命令から再び実行されます．

RESUMEステートメントのほかに，エラー処理ルーチンから実行を戻すのにRESUME NEXTステートメントもあります．これは，エラーを引き起こした次のステートメントに実行を移すもので，RESUMEステートメントとは必要に応じて使い分けます．もうひとつRESUME行ラベル（または行番号）というステートメントもありますが，この行ラベルはSUBあるいはFUNCTIONプロシージャにあってはいけないため，使い方がかなり限定されます．普通は使わないほうがよいでしょう．

イベントトラッピングは，ある特定のキーが押された場合などに，プログラムの流れを中断して，あらかじめ用意しておいたイベント処理サブルーチンに実行を分岐させます．“イベント”には，ある特定のキーが押された場合のほかに，システムクロックの経過時間，ライトペンあるいはジョイスティックの操作，シリアルポート通信，PLAYコマンドによる音楽のバッファの変化などがあります．イベントトラッピングを行うにはまず，

```
ON イベント GOSUB  行番号(行ラベル)
```

というステートメントを書いておきます．そして，イベントトラッピングを開始したい場所で，

```
イベント ON
```

ステートメントを書きます．イベントトラッピングを中止させたい場合はその位置で，

```
イベント OFF
```

ステートメントを書けばよく，再びイベント ONステートメントを書くまでイベントトラッピングは働きません．

イベント処理サブルーチンをモジュールレベルコードに書かなければいけないのは，エラートラッピングの場合と同様で，これらのトラッピングを使いにくい

ものにしています．ただし，複数のモジュールからなるプログラムの場合，メインモジュールのモジュールレベルコードに書かれたイベント処理サブルーチンを，すべてのモジュールのプロシージャから利用できます．

イベントトラッピングの例のひとつとして，**2.3**のパーソナル単語帳プログラムで使った，タイマートラッピングを説明しましょう．

ここでは，システムクロックを使って，パーソナル単語帳のディスプレイ画面に現在時刻を1秒単位で表示させて，時計の働きもさせようというものです．

メインモジュールのモジュールレベルコードにON イベント GOSUB 行番号 ステートメントを書きます．ここでは，

```
ON TIMER(1) GOSUB DispTime
```

と書かれています．これは，TIMERの引数が示す値の秒数ごとに，サブルーチンDispTimeの処理を行います．例では引数が1ですから1秒ごとにサブルーチンの処理を行います．

次に，メインモジュールのモジュールレベルコードに，イベント処理サブルーチンDispTimeを記述します．

その内容は，まず，

```
XPos=POS(0)
YPos=CSRLIN
```

によって，処理を行う直前のカーソルのXとY座標の値を，変数XPosとYPosに保存します．

次に，

```
LOCATE ClockRow, ClockCol
PRINT DATE$
LOCATE ClockRow+1, ClockCol+1
PRINT TIME$
```

の命令でディスプレイの所定の位置に，年月日と時刻の表示を行います．

そして，

LOCATE XPos, YPos

で保存しておいたカーソル位置にカーソルを戻しておきます.

この処理を1秒ごとに行いますから，この一連の処理は，単語帳プログラムの処理とは独立して，時刻を画面に表示しているように見えるわけです.

これでタイマーイベントトラッピングの準備が整ったので，あとは，画面に時刻表示をさせたいところで

TIMER ON

を，表示させたくないところで

TIMER OFF

を記述すればよいわけです．パーソナル単語帳プログラムでは，基本的にどの画面でも時刻表示を行うこととし，2，3の不用な場面のみTIMER OFFを使うことにしています．リスト1.4.4にサブルーチンDispTimeのコードを示しておきます.

●リスト 1.4.4

```
    :
ON TIMER(1) GOSUB DispTime
    :
DispTime:

    XPos = POS(0)
    YPos = CSRLIN

    LOCATE ClockRow, ClockCol
    PRINT DATE$
    LOCATE ClockRow + 1, ClockCol + 1
    PRINT TIME$

    LOCATE YPos, XPos

RETURN
```

1.4.5 モジュール化のための留意点

プログラムをモジュール化することは，たしかにメリットも多く，第2部のいくつかのプログラムでも，モジュールに分割したプログラム構成にしています．しかし，モジュール化をした場合，いろいろな注意事項を怠ってプログラミングを行うと，思わぬバグを発生させることになります．そしてそのバグもプログラムが分割されているがゆえに，発見が難しく，プログラム作成効率をかえって悪化させるといった事態を引き起こす場合もでてきます．

そこでここでは，モジュール化をする場合に，とくに留意しなければいけない点を考えてみたいと思います．

◉モジュール間の値の引き渡しは引数渡しで

複数のモジュールからなるプログラムで，全モジュールで特定の変数を共有することはできますが，これは各モジュールの独立性をかなり犠牲にするので，できるだけ避けたほうがよいでしょう．とくに数多くの変数を共有させるのはバグ発生のもとです．それぞれのプロシージャ間で引数渡しにすれば，モジュールの独立性も保て，モジュール化のメリットも増すと思われます．

◉CONSTの上手な使用法

QBではCONSTが使えるようになり，ずいぶんプログラムの可読性がよくなりました．定数をうっかり書き換えてしまうというトラブルもなくなるでしょう．ところが，CONSTはひとつのモジュール内だけで有効なため，プログラムを構成するほかのモジュールで宣言をし忘れると，そちらでは変数として扱われてしまい，思わぬバグを発生してしまいます．このトラブルを防ぐには，共通に使う定数を忘れないように，すべてのモジュールで宣言することですが，定数の数が多いと漏れが生ずる恐れがあります．そこで，メインモジュールのモジュールレベルコードでCONSTを宣言する場合，なるべくまとめて宣言して，サブモジュールにはそれをまるごとコピーするようにすればトラブルは防げます．

また，もうひとつの方法として，$INCLUDEメタコマンドを使用する方法があ

ります．これは，CONSTの宣言をまとめて行ったら，それらだけを，ファイル名をつけてフロッピィディスクにセーブしておきます．

たとえば，次のような記号定数を複数のモジュールで使いたい場合，この2行をフロッピィディスクにテキストファイルとしてセーブしておきます．

```
CONST PI = 3.14159
CONST False = 0, True = NOT False
```

ここでは，仮に“TEISU1.BAS”というファイル名でセーブしたとします．

次に，各モジュールでCONST文を記述する代わりに，次のように，$INCLUDEメタコマンドを記述すればよいのです．

```
'$INCLUDE: 'TEISU1.BAS'
```

QBは，フロッピィディスクからファイル“TEISU1.BAS”をインクルードしてきて，メタコマンドと置き換えます．この方法を使えば，いくつかの定数を宣言し忘れることによって起こる，比較的発見しにくいバグの発生を防止することができます．

1.5 リスト

情報x_1，x_2，…，x_nをまとめて(x_1，x_2，…，x_n)のように並べたものを線形リスト，x_1，x_2，…，x_nを要素と呼びます．要素x_iに対してx_{i-1}を左側の要素，x_{i+1}を右側の要素と呼びます．要素を1個も持たない線形リスト( )を空な線形リストと呼びます．以下，本章で扱うリストは線形リストなのでそれを単にリストと呼ぶことにします．

1.5.1 1次元配列上のリスト

情報x_1，x_2，…，x_nを順に並べる方法には，たとえば配列x(i)(i=1,2,…n)に順にx_1，x_2，…，x_nを入れることが考えられます．x_1，x_2，…，x_nが昇順に並んでいるとすると，配列x( )に実現されたリストは，その大きさが昇順に並んだものとなります．したがって，x(i)の値は，小さいほうからi番目の値となります．

ところで，配列のような，はじめから順に並んでいるメモリに情報を格納すると，新しい情報yを加えようとするとき，その順序を崩さないようにyをリストに付加するには，yより大きな値を持つ情報を，すべて1だけ右にずらさなくてはなりません．情報の数nが小さい場合はともかく，大きくなるとこれはなかなかやっかいな話になります．

逆に，情報x_iを除くとすると，穴をうめるためにやはりx_{i+1}，…，x_nを左にすべて移動しなくてはなりません．要素の数nが大きくなると，移動に要する時間も多くなるので，この形式でリストを実現することはあまりありません．

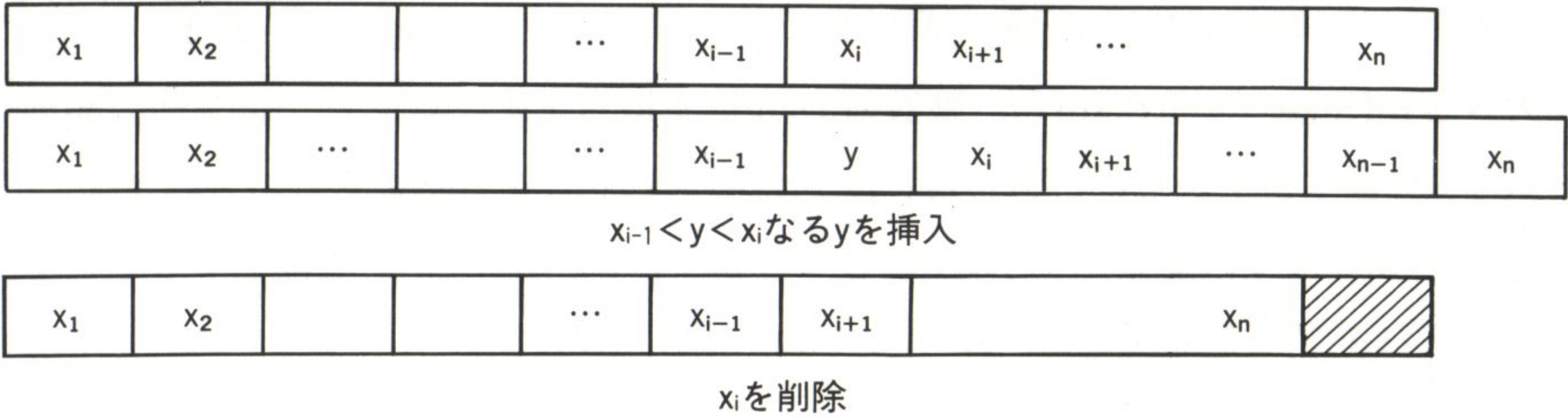
●図1.5.1 配列上に実現したリストと，リストへの挿入および削除

x_1	x_2			…	x_{i-1}	x_i	x_{i+1}	…	x_n

x_1	x_2	…		…	x_{i-1}	y	x_i	x_{i+1}	…	x_{n-1}	x_n

$x_{i-1}<y<x_i$なるyを挿入

x_1	x_2			…	x_{i-1}	x_{i+1}	x_n	

x_iを削除

1.5.2 リストのリンク表現

線形リストに対して新しい要素を加えたり，逆に要素を除去したりする操作が必要な場合には，１次元配列を利用したリストは使いやすくありません．このような場合には，要素を含む記憶領域の相互の結合を表すことのできる，リンク表現と呼ばれる方法を用います．

リンク表現では，要素の情報を格納するための記憶領域のほかに，結合を表すリンク情報を格納する記憶領域を必要とします．１個の要素の情報と，リンク情報をまとめて格納する記憶領域を節(node)と呼びます．節の中に，左と右のリンクを同時に含むものを２方向リンク表現，１方向のリンクのみを含むものを１方向リンク表現と呼びます．

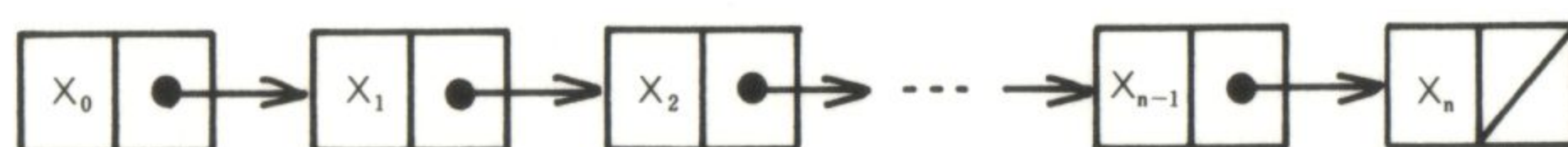
QBでは，TYPE宣言により，データ構造を表現するデータ型を作成可能です．１方向リンク表現を実現する例として，次のような型宣言と変数宣言をしておきます．

```
CONST ListSize=100
CONST InfSize=20
TYPE node
        Inf As STRING * InfSize
        right As INTEGER
END TYPE

DIM x(ListSize)As node
```

ここに，ListSize＝100とは，節の数が最大100個であること，InfSize＝20は，格納すべき情報が最大20文字以内であることを表します．型nodeは，要素として，大きさInfSizeバイトのInfと，整数型のrightを持ちます．そして，node型の配列x( )を宣言します．この宣言では，配列の添字はゼロから始まるものとします．

型nodeは２つの要素Infとrightからなります．rightはリンク情報を持っています．これを２つの部屋を持つ箱で表します(図1.5.2)．

●図1.5.2 1方向リンク表現リスト

X_0 → X_1 → X_2 → … → X_{n-1} → X_n ⧄

箱の中の左の部屋を情報が格納されたInfに，右の部屋をリンクを表すrightに対応させてあります．図1.5.2のリストの一番左の節は特別な節とし，リストの先頭にリンクするものとします．プログラム上ではx(0).rightとします．その節には情報は格納しません．一番右の節のリンクは，これ以上存在しないので，⧄で表します．プログラム上は，定数Nilをゼロと宣言し，Nilを書き込んでおくことにします．各x( )の添字は，配列の添字とは普通は一致しません．

簡単にするために要素としてPascal，Modula2，C，MBASIC，QuickBASICの５つの語を配列x( )に格納し，リストを構成してみます．

はじめは，どの節にも情報は入っていないので，次のようになります．

```
x(0).right=Nil
```

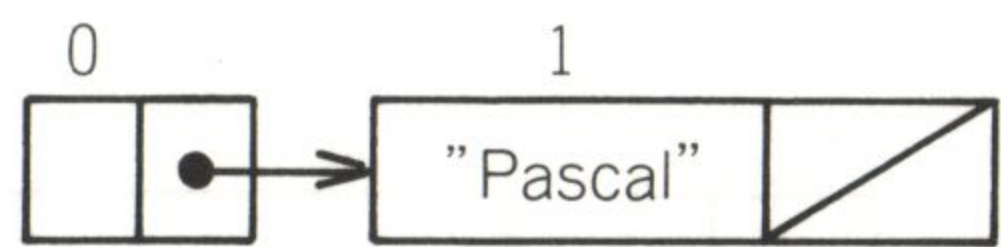
図1.5.2で言えば，一番左の箱のみが存在し，その節の右の部屋は⧄となっています．

最初の情報として，Pascalをリストに加えます．これをx(1).Infに格納すると，次のようになります．

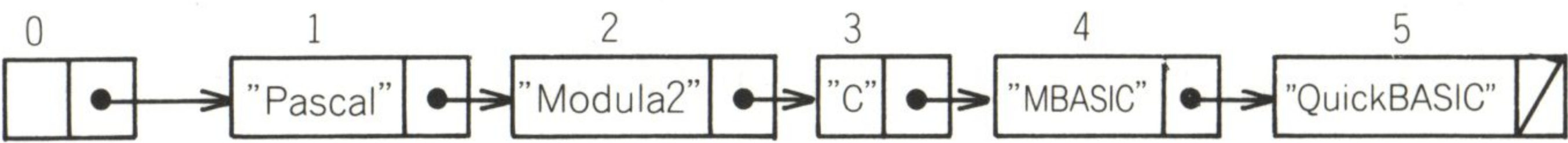
```
x(1).Inf="Pascal"
x(1).right=Nil
x(0).right=1
```

図で描けば，図1.5.3のようになります．

●図1.5.3

0 1
"Pascal"

以下同様にして,Modula2, C, MBASIC, QuickBASICを書き込みます. まだ使用されていないx(2), x(3), x(4), x(5)に順に入れるとします. すると,

●図1.5.4

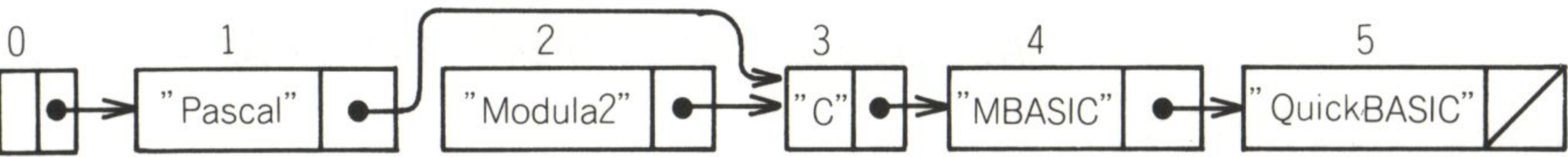
0 1 2 3 4 5
"Pascal" "Modula2" "C" "MBASIC" "QuickBASIC"

となります. 左の箱の上の数字は, 配列の添字を表します.

この図を見ると,図1.5.1の1次元配列を使用したリストとなんら変わるところがないように感じられます. しかし, リストからある箱を削除することを考えるとこの意味がはっきりします. たとえば, 上のリストから "Modula2" を削除するとします. この場合には, 単にリンクを書き換えればよいことになります. すなわち, "Pascal"を格納するx(1)の右リンクを, 直接3とすればよいのです. プログラム的には,

```
x(1).right=3
```

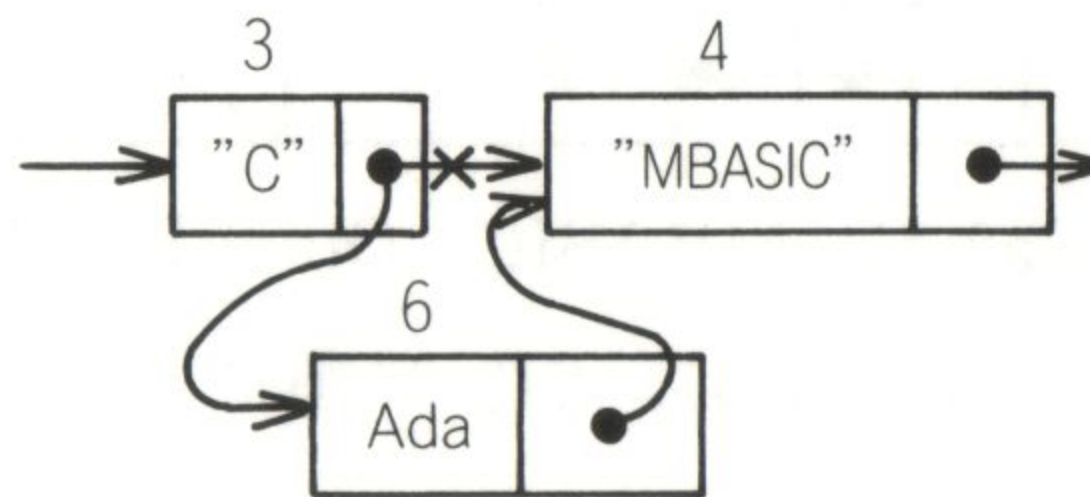
とすれば, 節x(2)は削除されたことになります. 図で描けば,

●図1.5.5

0 1 2 3 4 5
"Pascal" "Modula2" "C" "MBASIC" "QuickBASIC"

となります.

新しい情報をリストに追加する場合には,次のようになります.x(6).Infに"Ada"を書き込み, それを "C" と "MBASIC" の間に入れたいとします. リンク部を書き換えることになります.

```
x(6).right=x(3).right
x(3).right=6
x(6).Inf="Ada"
```

●図1.5.6

注意すべきは，はじめの2つの代入文の順序は逆にできないことです．

以上述べた内容を実行するプログラムをリスト1.5.1に示します．各プロシージャは次のように働きます．

SUB AddNode(a$,i)

x(i).Infにa$を書き込んで，リストのあとに接続します．

SUB DeleteNode(i,j)

x(i)のあとの節x(j)を除きます．

SUB InsertNode(a$,i,j)

a$をx(j).Infに書き込み，x(i)のあとに挿入します．

SUB PrintNode

リストの内容を節の添字とともに表示します．

また，変数x( )は，グローバルとするためにSHAREDで宣言しておきます．

以上で，リストのリンク構造と最小限の機能である節の挿入と削除について，理解されたと思います．

実際にプログラムするためには，もう少しいろいろなことを考える必要があります．そのひとつは，削除されたために，いわゆるゴミとなるメモリの処理です．ここでは，削除された節を，再利用可能なリスト(それを自由リストと名づける)に格納することにします．したがって，メモリ上には，“Pascal”，“C”などの情報の書き込まれたリストと，情報が書き込まれていないか，あるいは不要となった情報からなる自由リストの2つを管理します．したがって，プログラム実行時には，この2つのリストが存在することになります．

次に，リストにデータを挿入する場合，昇順あるいは降順にリストを構成することにします．挿入すべき位置を自動的に捜し，その位置に書き込みます．たと

●リスト 1.5.1

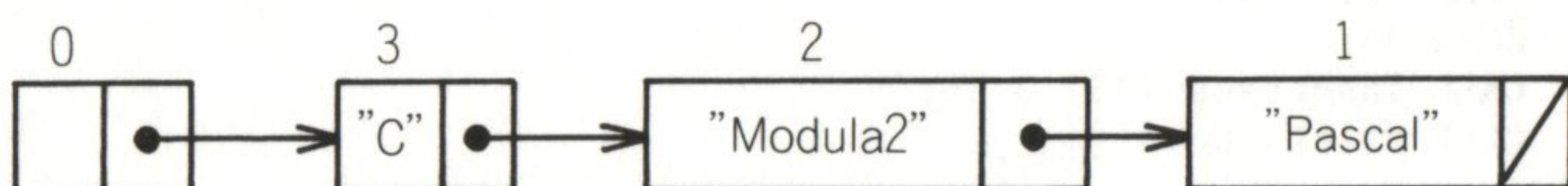
```
 1:   DECLARE SUB InsertNode (a$, i!, j!)
 2:   DECLARE SUB AddNode (a$, i!)
 3:   DECLARE SUB DeleteNode (i!, j!)
 4:   DECLARE SUB PrintNode ()
 5:   CONST ListSize = 100
 6:   CONST InfSize = 20
 7:   CONST Nil = 0
 8:
 9:   TYPE node
10:       Inf    AS STRING * InfSize
11:       right    AS INTEGER
12:   END TYPE
13:
14:   DIM SHARED x(ListSize)    AS node
15:
16:   DATA Pascal,Modula2,C,MBASIC,QuickBASIC,Ada
17:
18:       x(0).right = Nil
19:       FOR i = 1 TO 5
20:           READ a$
21:           CALL AddNode(a$, i)
22:       NEXT i
23:       CALL PrintNode
24:       CALL DeleteNode(1, 2)
25:       CALL PrintNode
26:       READ a$
27:       CALL InsertNode(a$, 3, 6)
28:       CALL PrintNode
29:
30:
31:   END
32:
33:
34:
35:   SUB AddNode (a$, i)
36:   '    x(i)...Added Node
37:       NextNode = 0
38:       DO
39:           IF x(NextNode).right = Nil THEN
40:               EXIT DO
41:           END IF
42:           NextNode = x(NextNode).right
43:       LOOP
44:       x(NextNode).right = i
45:       x(i).right = Nil
46:       x(i).Inf = a$
47:   END SUB
48:
49:   SUB DeleteNode (i, j)
50:   '    x(j)...Deleted Node
51:   '    x(i)...preceeding Node
52:       x(i).right = x(j).right
53:   END SUB
54:
55:   SUB InsertNode (a$, i, j)
56:   '    x(j)...Inserted Node
57:   '    x(i)...Preceeding Node of x(j)
58:
59:       x(j).right = x(i).right
```

```
60:         x(i).right = j
61:         x(j).Inf = a$
62:
63:     END SUB
64:
65:     SUB PrintNode
66:        PRINT
67:         IF x(0).right = Nil THEN
68:             EXIT SUB
69:         END IF
70:         NextNode = x(0).right
71:         DO
72:             PRINT USING "###  @"; NextNode; x(NextNode).Inf
73:             NextNode = x(NextNode).right
74:         LOOP UNTIL NextNode = Nil
75:     END SUB
76:
```

えば“Pascal”，“Modula2”，“C”をこの順に挿入すると，次の図のリストができます．

●図1.5.7

0　3　2　1

"C"　"Modula2"　"Pascal"

このリストに，データ“MBASIC”を挿入するには“C”のあとに入れなくてはなりません．“C”はx(3)に入っているので，添字3を捜す必要があります．x(0).rightがリストの先頭を指しているので，リストの先頭から，挿入したいデータより大きいか等しいデータを含む節を順に捜します．その節を指している節の添字が，求める添字になります．プログラムで表すと次のようになります．a$に挿入したいデータが入っているものとします．

```
j=0
i=x(j).right
DO
   IF  a$<=x(i).Inf THEN
          EXIT DO
   ELSEIF x(i).right=Nil THEN
          j=i
          EXIT DO
```

```
    ELSE
      j=i
      i=x(i).right
    END IF
  LooP
  FindInsertedPos=j
```

実際のプログラムでは，上のルーチンを関数にしてあります．

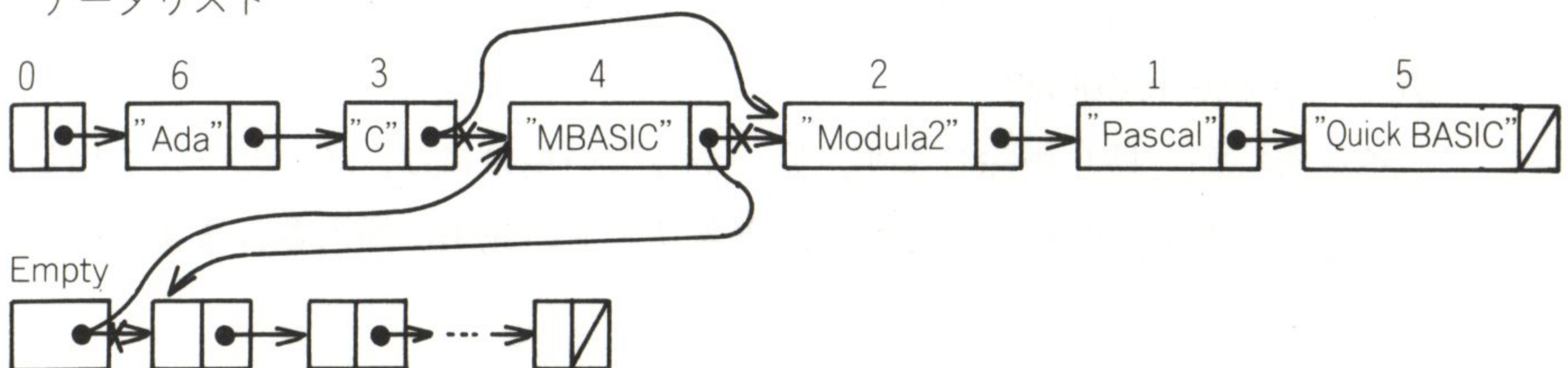
次に，削除する場合を考えます．削除したいデータを含む節の添字を捜すためには，上の方法と類似の考え方をします．ただし，不等号は等号にし，次がリストの終わりになったら終了とします（すなわちリスト中に削除したいデータは存在しなかった）．この部分は全体のプログラム**リスト1.5.2**を参照してください．

削除された節（配列x( )）は，再使用のために自由リストに加えます．自由リストの先頭を指す変数をEmptyとすると，

```
x(i).right=Empty
Empty=i
```

となります．ここに，iは削除された節の添字です．図で示すと次のようになります．情報の入っているリストをデータリストと名づけます．そこから，“MBASIC”を削除します．

“MBASIC”を含むx(4)がデータリストから削除され，自由リストの先頭に加えられました．逆に，自由リストの利用のためには，自由リストの先頭の節を取り出します．挿入に先立ち自由リストの有無が調べられ，可能な節があるとき，その添字を返すには次のようにします．

●図1.5.8

```
IF Empty=Nil THEN
      PRINT"NOT available any Free node."
      GetNode=Nil
ELSE
      GetNode=Empty
      Empty=x(Empty).right
END IF
```

上に述べた内容を実際にプログラムしたものをリスト1.5.2に示します．ただし，プロシージャPrintNodeは，リスト1.5.1のそれと同一なので，リスト中からは省いてあります．

各プロシージャについて簡単に説明しておきます．

SUB InitFreeList

プログラムのはじめに実行され，すべての配列を自由リストに加えます．リストの先頭は，変数Emptyが指します．

FUNCTION FindInsertedPos(a$)

a$を挿入すべき位置を返す変数です．関数値を添字とする節のあとに挿入します．

プロシージャSUB InsertNode(a$)中で呼ばれます．

FUNCTION GetNode

自由リストの先頭の位置を返します．自由リストが空の場合には，Nilを返します．プロシージャSUB InsertNode(a$)中で呼ばれます．

SUB InsertNode(a$)

a$を，左から右へと昇順に並べたリストの該当する位置に挿入します．

SUB DeleteNode(a$)

a$を含む節を捜し削除します．見つからない場合には，メッセージを出して終了します．削除された節は，空きリストの先頭に挿入されます．

●リスト 1.5.2

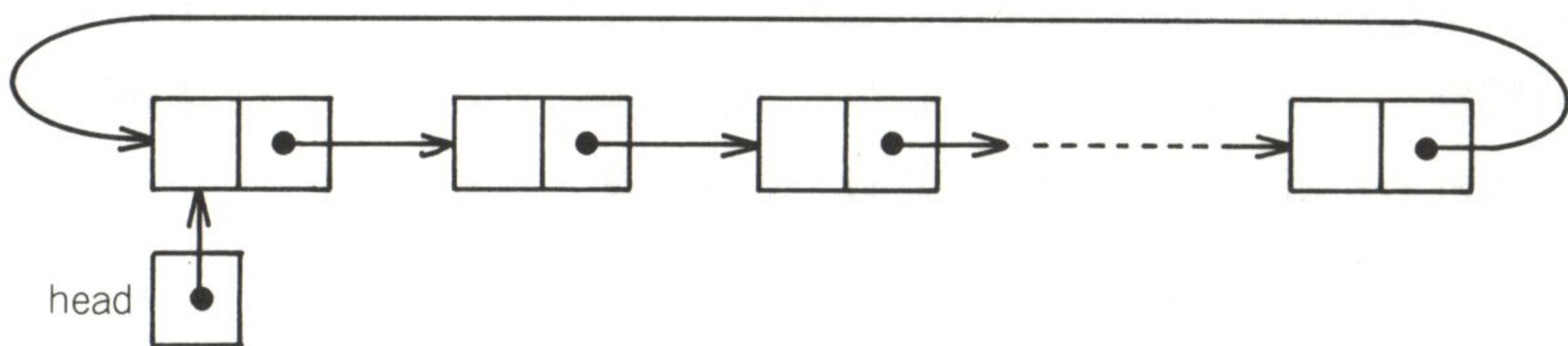
```
 1:  DECLARE FUNCTION FindInsertedPos! (a$)
 2:  DECLARE SUB InitFreeList ()
 3:  DECLARE FUNCTION GetNode! ()
 4:  DECLARE SUB InsertNode (a$)
 5:  DECLARE SUB DeleteNode (a$)
 6:  DECLARE SUB PrintNode ()
 7:
 8:  CONST ListSize = 6
 9:  CONST InfSize = 20
10:  CONST Nil = 0
11:  CONST false = 0, true = NOT false
12:
13:   TYPE node
14:       Inf     AS STRING * InfSize
15:       right     AS INTEGER
16:   END TYPE
17:
18:   DIM SHARED x(ListSize)     AS node
19:   DIM SHARED Empty
20:
21:   DATA Pascal,Modula2,C,MBASIC,QuickBASIC,Ada
22:
23:       x(0).right = Nil
24:       CALL InitFreeList
25:       FOR i = 1 TO 6
26:           READ a$
27:           CALL InsertNode(a$)
28:           CALL PrintNode
29:           INPUT z$
30:       NEXT i
31:
32:       RESTORE
33:       FOR i = 1 TO 6
34:           READ a$
35:           CALL DeleteNode(a$)
36:           CALL PrintNode
37:           INPUT z$
38:       NEXT i
39:
40:   END
41:
42:   SUB DeleteNode (a$)
43:       '   x(i)...Deleted Node
44:       '   x(j)...preceeding Node
45:       b$ = LEFT$(a$ + SPACE$(InfSize), InfSize)
46:       j = 0
47:       i = x(j).right
48:       DO
49:           IF b$ = x(i).Inf THEN
50:               EXIT DO
51:           END IF
52:           j = i
53:           i = x(j).right
54:      LOOP UNTIL i = Nil
55:       IF i = Nil THEN
56:           PRINT "Not found "; CHR$(34); a$ + CHR$(34); " in this List."
57:           EXIT SUB
58:       END IF
59:
60:       x(j).right = x(i).right
```

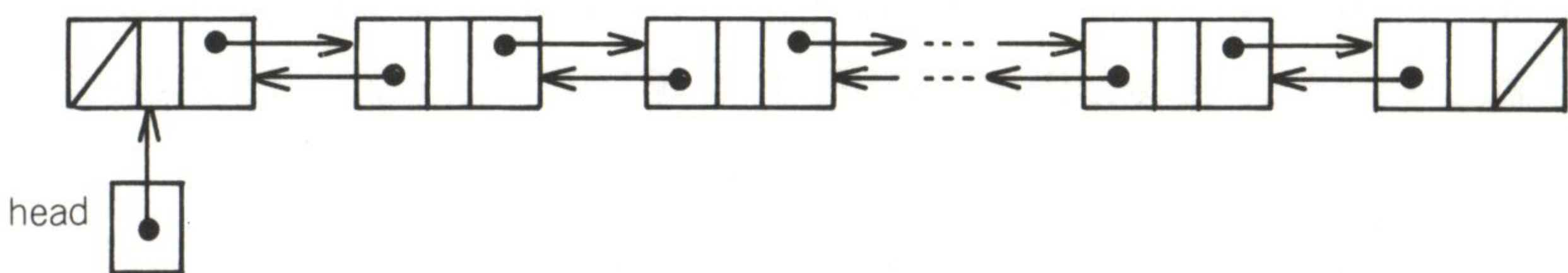
```
 61:         x(i).right = Empty
 62:         Empty = i
 63:
 64:     END SUB
 65:
 66:     FUNCTION FindInsertedPos (a$)
 67:         j = 0
 68:         i = x(j).right
 69:         b$ = LEFT$(a$ + SPACE$(InfSize), InfSize)
 70:         DO
 71:             IF b$ <= x(i).Inf THEN
 72:                 EXIT DO
 73:             ELSEIF x(i).right = Nil THEN
 74:                 j = i
 75:                 EXIT DO
 76:             ELSE
 77:                 j = i
 78:                 i = x(i).right
 79:             END IF
 80:         LOOP
 81:         FindInsertedPos = j
 82:
 83:     END FUNCTION
 84:
 85:     FUNCTION GetNode
 86:         IF Empty = Nil THEN
 87:             PRINT "Not available any free node."
 88:             GetNode = Nil
 89:             EXIT FUNCTION
 90:         END IF
 91:         GetNode = Empty
 92:         Empty = x(Empty).right
 93:
 94:     END FUNCTION
 95:
 96:   SUB InitFreeList
 97:         i = 1
 98:         DO
 99:             x(i).right = i + 1
100:             i = i + 1
101:         LOOP UNTIL i > ListSize
102:         x(ListSize).right = Nil
103:         Empty = 1
104:   END SUB
105:
106:     SUB InsertNode (a$)
107:       '   x(i)...Preceeding Node of x(j)
108:       '   x(j)...store a$
109:
110:         i = FindInsertedPos(a$)
111:         j = GetNode
112:         IF j <> Nil THEN
113:             x(j).right = x(i).right
114:             x(i).right = j
115:             x(j).Inf = a$
116:         END IF
117:
118:     END SUB
```

1.5.3 そのほかのリスト表現

前節では，左端を先頭とする１方向リンク表現リストを説明しました．リストの先頭は，x(0).rightが指しています．この場合には，添字ゼロの節を，先頭を指すヘッダとして利用しました．一般的には，たとえば変数headを利用したほうが挿入や削除のプログラムはいくらか複雑になりますが，可読性はよくなるでしょう．

線形リストには，図1.5.2で示した直線状にデータの並んだもののほかに，右端が左端を指す環状のものもあります．

●図1.5.9 環状リスト

また，リンクが左右両方向を指すものもあります．線形リストは下図のように表されます．

●図1.5.10 2方向性リスト

このリストの場合には，添字iを持つ節の削除は，節が右端または左端でないとき，

```
x(x(i).right).left=x(i).left
x(x(i).left).right=x(i).right
```

と簡単になります．**1.5.2**で述べたように，x(i)を指す節の添字jを必要とするようなことはありません．

1.6 再帰

再帰とは，自らのもとに帰ることを意味し，プログラム上では，再帰的呼び出しという形で現れます．プログラムが再帰であるためには，そのもとのアルゴリズムが再帰であることが必要です．アルゴリズムが再帰とならないプログラムに，再帰的呼び出しを用いることはできませんが，処理速度やスタックの深さの制約から，再帰的アルゴリズムを非再帰的に変更することはよく行われます．

従来のBASICでは，GOSUBによるサブルーチンコールを用いて再帰的に行うことが可能でしたが，変数がすべて大域的であるために，再帰的呼び出しを行うためには特別な工夫が必要でした．しかし，QBでは，プロシージャの再帰的呼び出しが許されるため，プログラミングはたいへん楽になりました．

1.6.1 再帰的呼び出し

再帰的呼び出しとは，プロシージャP内で再びPを呼び出すことです．より一般的には，２つ以上のプロシージャP_1，P_2，…，P_nがP_1の中でP_2を，P_2の中でP_3，…と順に呼び出し，最後にP_nの中で再びP_1を呼び出すようなものも再帰的呼び出しと言います．

再帰的呼び出しでよく知られているものに階乗のプログラミングがあります．階乗は次のように定義されます．

$$n! = \begin{cases} 1 & n=0 \\ n(n-1)! & n>0 \end{cases}$$

nがゼロでないかぎり，nと(n－１)！の積を求めます．これをプログラムすると，リスト1.6.1のようになります．上式が，そのまますなおにプログラムされています．

●リスト 1.6.1 階乗のプログラム

```
  1: DECLARE FUNCTION fact& (n&)
  2: 'Factorial demo
  3:
  4: DIM n AS LONG
  5: DIM f AS LONG
  6:
  7:     FOR n = 1 TO 12
  8:         f = fact&(n)
  9:         PRINT "Factorial("; n; ")="; f
 10:     NEXT n&
 11: END
 12:
 13:
 14: FUNCTION fact& (n&)
 15:     IF n& <= 0 THEN
 16:         fact& = 1&
 17:     ELSE
 18:         fact& = n& * fact&(n& - 1)
 19:     END IF
 20: END FUNCTION
 21:
```

再帰的呼び出しプログラミングで注意することは，呼び出しの停止条件を忘れないことです．この例では，引数の値がゼロの場合，定数1を関数値としてとり，それ以上自分自身を呼び出さず，再帰呼び出しは終わります．

再帰的呼び出しを含むプログラムの例を2つあげます．

【例1】リストの逆順出力

前節で作成されたリストの内容を逆順に出力するプログラムをリスト1.6.2に示します．関数PrintByReverseは引数として，リストの先頭，すなわち，x(0).rightを渡して呼ぶと，リストの節の内容を逆順にプリントします．

x(i%).rightがNilでないときは，そのときの節の内容を変数sに覚えておき，その右リンクされた節を調べます．その先にリストの要素がない，すなわち，パラ

●リスト 1.6.2 逆順出力プログラム

```
113:
114: SUB PrintByReverse (i%)
115: DIM s AS STRING * InfSize
116:     IF x(i%).right <> Nil THEN
117:         CALL PrintByReverse(x(i%).right)
118:     END IF
119:     s = x(i%).inf
120:     PRINT USING "### @"; i%; s
121: END SUB
```

メータi=Nilになると，それ以上の呼び出しをやめ，セーブされていた内容をプリントします．最後に呼び出されたときの節の内容が1番目に，その直前に呼ばれた節の内容が2番目に，…と逆順にプリントされます．

【例2】最大公約数〔GCD〕

2数aとbの最大公約数は，a＞bとすると，aをbで割った余りrがゼロならば，そのときのbが最大公約数，もし，ゼロでないならば，r＜bなので，bとrについて同様な手続きを繰り返すことにより得られます．

これは，r＜bの場合，bをaに代入し，rをbに代入することにより，再び同一のアルゴリズムが使用されるので，再帰的アルゴリズムと言えます．

リスト1.6.3にプログラム例を示します．このリストでは，2つの引数は第1番目のほうが第2番目より等しいか大きくないと正しく動作しません．原理のみを示すために，引数のチェックは省略してあります．

aをb(a＞b)で割ることは，aからbを，差がbより小さくなるまで減じることを意味します．したがって，次のようにも表されます．

a＝b　ならば　GCD＝a

a＞b　ならば　GCDはa－bとbのGCD

b＞a　ならば　GCDはaとb－aのGCD

上の表現で，a＝bとは，aをbで割って余りがゼロを表します．

この表現によるプログラム例をリスト1.6.4に示します．

●リスト 1.6.3　GCDのプログラム(その1)

```
 1: DECLARE FUNCTION GCD! (a!, b!)
 2:     x = 1350: y = 405
 3:     PRINT "GCD(x"; x; ","; y; ") = "; GCD(x, y)
 4:  END
 5:
 6: FUNCTION GCD (a, b)
 7:     r = a MOD b
 8:     IF r <> 0 THEN
 9:         GCD = GCD(b, r)
10:     ELSE
11:         GCD = b
12:     END IF
13: END FUNCTION
14:
```

●リスト 1.6.4 GCDのプログラム(その2)

```
 1: DECLARE FUNCTION GCD! (a!, b!)
 2:  'GCD sample
 3:
 4:     x = 1350: y = 405
 5:     PRINT "GCD("; x; ","; y; ")="; GCD(x, y)
 6:  END
 7:
 8: FUNCTION GCD (a, b)
 9:
10:     IF a = b THEN
11:         GCD = a: EXIT FUNCTION
12:     ELSEIF a > b THEN
13:         GCD = GCD(a - b, b)
14:     ELSE
15:         GCD = GCD(a, b - a)
16:     END IF
17:
18: END FUNCTION
19:
```

1.6.2 クイックソート

ソート(sort)とは，整列とも呼ばれ，データを大きさの順に並べることです．クイックソート(Quick sort)とは文字どおり高速なソートの意味で，ランダムな順に並べられたデータをソートする場合に高速で便利な方法です．

クイックソートは,アルゴリズムが再帰であることによっても知られています．その基本的な考え方は次のとおりです．

ソートすべきn個のデータの中のある値を取り出す．その値をMとする．Mより小さな値をMの左側に，大きな値をMの右側に並べる．この段階で，Mの左側には，Mより小さなランダムに並んだデータ列，右側にはMより大きなランダムに並んだデータ列が得られる．そこで，そのおのおのの右側と左側の区間について,同じ考えで区間を分割する.区間の幅が1になったら終了する(図1.6.1)．

これを，もう少し具体的に記すと，次のようになります(J. L. ベントリー，コラム10による)．

データは配列x(l)，x(l+1)，…，x(u−1)，x(u)に入っているとします(第1回目はl=1,u=nです)．

●図1.6.1 クイックソートの考え方

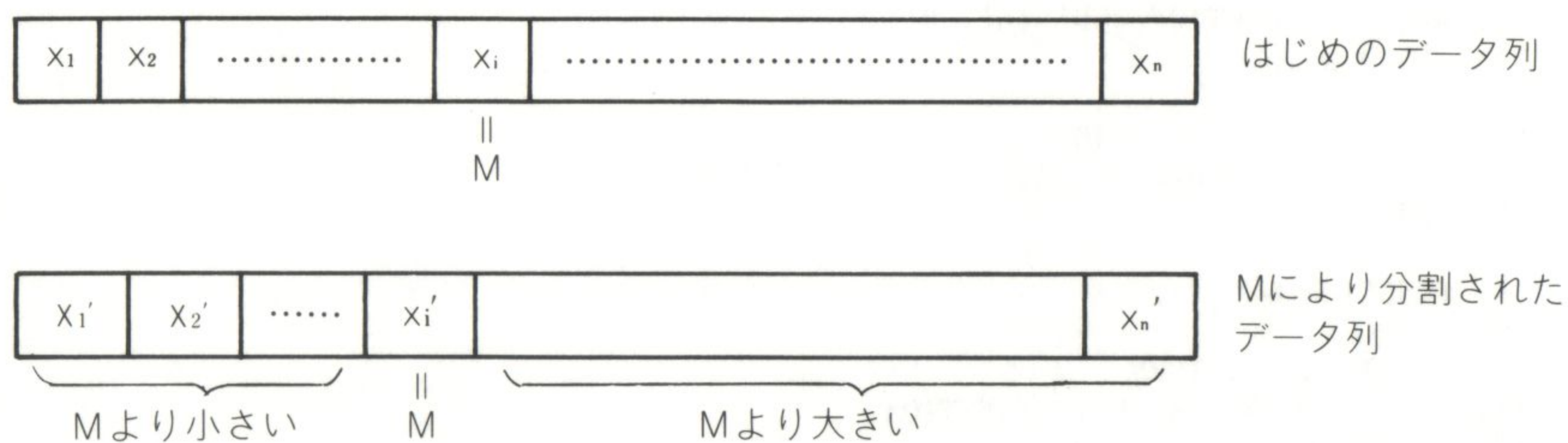
x_1	x_2	………………	x_i	…………………………………………	x_n

はじめのデータ列

x_i ＝ M

x_1'	x_2'	……	x_i'		x_n'

Mにより分割されたデータ列

Mより小さい　x_i' ＝ M　Mより大きい

もし，l ＝uならば，１個しかデータがないので，“ソート済み”と考えます．したがって，l ＜uの場合にのみ，以下の手順(1)～(4)が実行されます(すなわち，l ≧uが再帰的呼び出しの停止条件です)．

（１）　x(l)，…，x(u)の中の任意の値をMとします．そして，Mとx(l)を交換します．そして，k← l とします．

（２）　i＝ l ＋１からuの間，以下の処理を実行します．
x(i)＜Mであるならば，kをインクリメントし，x(i)とx(k)を交換します．すなわち，k＞ l ならば，添字kを持ったデータx(k)は，Mより小さいことになります．

（３）　左端におかれていた値Mとx(k)を交換します．これにより，添字がk－１以下の配列のデータはMより小さいものばかりになっています．添字がk＋１以上の配列のデータは，Mに等しいか大きなものになっています（(２)が実行されない場合，すなわちk＝ l のままの場合は交換する必要はありませんが，そのようなことはめったにないので，交換しても時間の無駄にはならないでしょう)．

（４）　左の区間 l ～k－１について，同様な処理を実行します．
右の区間k＋１～uについて同様な処理を実行します．

数値例で確かめてみましょう．
はじめのデータが，x(1)～x(10)中に，

75　90　97　33　0　54　45　42　70　34

で与えられたとします．(１)でM＝45とすると，75と交換し，k＝１となります．

(2)を実行すると，左端の値45はそのままで，その右側に45より小さな値が並びます．

すなわち，

45 33 0 42 34 54 75 90 70 97

ここでk＝5となっています．

(3)を実行すると，

```
34  33  0  42    45    54  75  90  70  97
└────┬────┘          └───────┬───────┘
  左の区間                  右の区間
```

45より小さなデータは左に，大きいか等しいデータは右の区間に分かれています．そこで，左の区間について同様の処理を行い，右の区間についても同様な処理を行います．

プログラム例をリスト1.6.5に示します．(1)の任意の値を選ぶ方法としては，J. L. ベントリーのプログラムにならって，乱数で，区間内の任意の添字を選び，その配列の値を用いています．

疑似データの発生，データのプリントへのプロシージャも含まれていますので，以下に簡単に説明しておきます．

SUB GenerateData(x( ),n)

配列x( )にn個の乱数を書き込みます．乱数は0～999の整数値をとります．

SUB PrintData(x( ),n)

配列x( )の内容をスクリーン上に1データ8カラムで表示します．スクリーン1行に10個のデータが出力されます．なお，データは整数値を仮定します．

●リスト 1.6.5 クイックソート(再帰的呼び出し)

```
 1: DECLARE SUB Qsort2 (x!(), lower!, upper!)
 2: DECLARE SUB Qsort1 (x!(), l!, u!)
 3: DECLARE SUB GenerateData (x!(), n!)
 4: DECLARE SUB PrintData (x!(), n!)
 5: CLEAR , , 5000
 6: CONST n = 100
 7: DIM x(n)
 8:
 9:     CALL GenerateData(x(), n)
10:     CALL PrintData(x(), n)
11:     PRINT TIME$
12:     CALL Qsort2(x(), 1, n)
13:     PRINT TIME$
14:     CALL PrintData(x(), n)
15: END
16:
17: SUB GenerateData (x(), n)
18:     FOR i = 1 TO n
19:         x(i) = INT(1000 * RND)
20:     NEXT i
21: END SUB
22:
23: SUB PrintData (x(), n)
24:     PRINT
25:     FOR i = 1 TO n
26:         PRINT USING "    #####"; x(i);
27:     NEXT i
28: END SUB
29:
30: SUB Qsort1 (x(), l, u)   'By Recursive Call
31:     IF l < u THEN
32:         SWAP x(l), x(l + INT(RND * (u + 1 - l)))
33:         k = l
34:         M = x(l)
35:         FOR i = l + 1 TO u
36:             IF x(i) < M THEN
37:                 k = k + 1
38:                 SWAP x(k), x(i)
39:             END IF
40:         NEXT i
41:         SWAP x(l), x(k)
42:         CALL Qsort1(x(), l, k - 1)
43:         CALL Qsort1(x(), k + 1, u)
44:     END IF
45:
46: END SUB
47:
```

1.6.3 再帰性の除去

QBにおけるプロシージャの呼び出しは,呼び出しごとに引数やローカル変数の領域を確保するために，呼び出しに時間がかかります(これは，PascalやCでも同様で，以下の議論は同じようにあてはまります)．クイックソートにおいて，データ数が分割の結果小さくなっても，区間の大きさ1(この場合何もしない)においてさえ,プロシージャの呼び出しが行われ,かえって能率が悪くなります．クイックソートでは，区間の大きさがある程度以下になったら，ほかのソートに切り換えるなどの方法がとられます．

再帰的なアルゴリズムは，記述には適していますが，プログラミング上は，非再帰的呼び出し，すなわち，1回だけの呼び出しで実行できるようにプログラムを書き換えたほうがよい場合が多いです．そこで，再帰的アルゴリズムを非再帰に書き改める例を2，3示します．

再帰的呼び出し時には,呼び出し時のローカル変数はメモリ上(通常，スタックメモリ上)に保存されており,呼び出しから戻った時点でその値を使用することが可能です．再帰的呼び出しの代わりに，ループを利用して，自分自身を再実行するためには，ローカルな変数のうちあとで使用するものを保存しておかなくてはなりません．クイックソートのプログラムでそのようすを調べてみます．

リスト1.6.5のプログラムの主要部は次のように表されます．

```
SUB QSORT (x( ),l,u)
  IF l<u THEN
    1 kを分割点とするようにして左と右の2つの部分にわける
    2 CALL QSORT (x( ),1,k−1)
    3 CALL QSORT (x( ),k+1,u)
  END IF
END SUB
```

2で自分自身を呼び出すことは，あらためて2つの引数 l とuを，l とk−1に置き換えて，プロシージャの本体を実行することに相当します．再帰的呼び出し

では，QSORT(x( )，l，k−1)の引数lは呼び出しごとに値が更新されています．その点に注意して，2のCALL QSORT( )を除くには，2で，lを新しいlに，uを新しいuにして，QSORTを呼び出す代わりに，IFの前にGOTOで分岐することに相当します．再帰的呼び出しでは，lとuの値は呼び出しごとに保存されている必要があるので，これを配列にして保存します．

3のCALLは，2のCALLがこれ以上再帰的呼び出しをしなくなってはじめて実行されるので，IF l <u THEN～END IFブロックのあとに出ます．そして，やはり，k＋1とuを新しい値に書き換えて，IFの前にGOTOで分岐します．

分割点kは，新しいuまたはlを求めるのに必要とされるので，保存する必要があります．lとu，それにkを保存するために配列l( )，u( )，k( )を用い，呼び出しの深さに相当するポインタを変数spで表すと，次のようになります．

```
          SUB QSORT (x( ),lower,upper)
            sp=1  :  l(sp)=lower:u(sp)=upper
      again:
            IF l(sp)<u(sp) THEN
               1'：分割点kを求める．mの左側はx(k)より小さく，mの右側はx(k)
                   に等しいか大きい2つの列に分割される
                   k(sp)=k       'kをセーブする
               2'：sp=sp+1  :  l(sp)=l(sp-1)  :  u(sp)=k-1
                   goto again
            END IF
            IF sp<>1 THEN
               3'：sp=sp-1  :  l(sp)=k(sp)+1
                   goto again
            END IF
          END SUB
```

上のプログラムをよく見ると，l(sp)は，1’と2’のループ中では値が変わらないことがわかります．したがって，lは保存する必要はありません．また，IF～THENの部分は，goto againを含むのでループに書き直すことができます．実行条件がはじめに調べられることから，DO WHILEループを用いて次のように表されます．

```
    SUB QSORT (x( ),lower,upper)
    sp=1  :  l=lower : u(sp)=upper
  again:
      DO WHILE    l<u(sp)
        1' : kを得る
                k(sp)=k      'save  k
        2' : sp=sp+1  :  u(sp)=k-1
      LOOP
      IF sp< >1 THEN
        3' : sp=sp-1  :  l=k(sp)+1
             goto again
      END IF
    END SUB
```

さらにgoto againは，spが1の値をとるまで実行されることから，DO/LOOP UNTIL型のループに置き換えられます．コーディング例をリスト1.6.6に示します．

上の例で引数 l を保存する必要がなかったのは，次の理由によります．

連続した2つの再帰的呼び出し，

```
  2   CALL QSORT(x( ),  l , k−1)
  3   CALL QSORT(x( ), k+1,  u)
```

のうち，2のCALLでは，はじめの引数 l は戻ってから再使用されません．他方，2番目の引数k−1は，非再帰的なプログラムでは，引数uに代入されることになるため，3で正しいuの値を保持するためには，保存しなくてはなりません．kの値も，2の実行中変化するので保存しておく必要があります．

一般には，再帰的呼び出しの直後にEND SUBやEND FUNCTIONがくるものは，ローカル変数や引数の値を保持しなくてもよいので，それらの保存を考えなくてもよい分，非再帰的アルゴリズムへの変更は簡単になります．

●リスト 1.6.6　クイックソートプログラム(非再帰的呼び出し)

```
48: SUB Qsort2 (x(), lower, upper)  'By Non-recursive Call
49:     DIM u(20), k(20)
50:     sp = 1: l = lower: u(sp) = upper
51:
52:     DO  ' loop for CALL Qsort(x(), k+1, u)
53:         DO WHILE l < u(sp) ' loop for CALL Qsort(x(), l, k-1)
54:          '1'
55:             SWAP x(l), x(l + INT(RND * (u(sp) + 1 - l)))
56:             M = x(l): k = l
57:             FOR i = l + 1 TO u(sp)
58:                 IF x(i) < M THEN
59:                     k = k + 1: SWAP x(k), x(i)
60:                 END IF
61:             NEXT i
62:             SWAP x(l), x(k)
63:             k(sp) = k
64:           '2'
65:             sp = sp + 1: u(sp) = k - 1
66:          LOOP
67:           '3'
68:          sp = sp - 1: l = k(sp) + 1
69:
70:     LOOP UNTIL sp = 0
71:
72: END SUB
73:
```

1.6.4 [付]ソートプログラム

本章でクイックソートのプログラムがでてきたので，非再帰的アルゴリズムで表されるいくつかのソートプログラムを紹介しておきます．なお出典は，ヴィルトの「アルゴリズム＋データ構造＝プログラム」(日本コンピュータ協会) によります．

1.6.4.1 バブルソート

バブルソートとは，ソートするようすを図示すると，あたかもあわ(bubble)が底から上に上がってゆくように見えるので名づけられたソート法です．理解しやすいですが，実行速度は遅いので，データ数が少ない場合にしか用いられません．

配列x( )に格納されたn個のデータをソートするとします．次のようにします．

はじめにn個のデータすべてについて隣同士を比較し，添字の小さいほうが値が

大きければ交換します．これを，添字がnから２まで順に実行すれば，添字１に最も小さい値が入ります．プログラムで示せば，

```
FOR J=n TO 2 STEP-1
    IF x(j-1) > x(j) THEN SWAP x(j-1), x(j)
NEXT J
```

x(1)が最小値であることがわかったので，次は，nから３まで同様のことを実行します．すると，x(2)に，次に小さい値が入ります．このようにしてnまで操作すれば，昇順に並んだデータが得られます．

実際のプログラムを，リスト1.6.7に示します．

●リスト 1.6.7　バブルソート

```
32: SUB BubbleSort (x(), n)
33:     FOR i = 2 TO n
34:         FOR j = n TO i STEP -1
35:             IF x(j - 1) > x(j) THEN
36:                 SWAP x(j - 1), x(j)
37:             END IF
38:         NEXT j
39:     NEXT i
40: END SUB
```

1.6.4.2 インサーションソート

インサーション(insertion＝挿入)ソートは，カードを番号順に並べたり，レポート類を順に並べる方法に由来するソートの方法です．たとえば，６枚のカードが，

[8] [6] [3] [4] [1] [9]

の順に得られたとします．これを並べ替えるには(人間は，直観的に処理する部分がありますが，それは考えないとして)，次のようなやり方をします．６は８より小さいので，６を８の左側に置きます．これにより，

[6] [8] [3] [4] [1] [9]

となります．次に３を調べます．３は８と６よりも小さいので，一番左へ置きます．それによって，

3 6 8 4 1 9

となります．次に，４を３と６の間に入れます．

3 4 6 8 1 9

次に１を３の左に置きます．

1 3 4 6 8 9

最後に９は，その左のどれよりも大きいのでこのままです．以上で並べ替えが終わりました．

一般的に言えば，

> 左からm－１番目までは，ソート済みであるとして，m番目のデータを，その左の数字列の間の適当な位置に挿入する．挿入によって，その間のデータを１つずつ移動させる

となります．

実際のプログラムでは，大小比較とデータの移動を交互に行うという巧妙な方法をとっています．また，一番左の端まできたら，それより先には挿入する位置がないので，停止させるために，添字ゼロの変数を番兵として使用しています．これがないと比較がもう少し面倒になります．

プログラム例をリスト1.6.8に示します．

1.6.4.3 セレクションソート

この方法は，バブルソートと似たところがありますが，データの交換を毎回しない分だけ高速です．

考え方は単純で，n個のデータがあったとすると，はじめにn個の中の最小値を求め，それをx(1)に置きます．次に，２番目からn番号のn個のデータの最小値をx(2)に置きます．以下，これを繰り返すのみです．

プログラム例をリスト1.6.9に示します．

●リスト 1.6.8 インサーションソート

```
48: SUB InsertionSort (x(), n)
49:     FOR i = 2 TO n
50:         t = x(i)
51:         x(0) = t
52:         j = i - 1
53:         DO WHILE t < x(j)
54:             x(j + 1) = x(j)
55:             j = j - 1
56:         LOOP
57:         x(j + 1) = t
58:     NEXT i
59: END SUB
60:
```

●リスト 1.6.9 セレクションソート

```
69: SUB SelectionSort (x(), n)
70:     FOR i = 1 TO n - 1
71:         k = i
72:         t = x(i)
73:         FOR j = i + 1 TO n
74:             IF x(j) < t THEN
75:                 k = j
76:                 t = x(j)
77:             END IF
78:         NEXT j
79:         x(k) = x(i)
80:         x(i) = t
81:     NEXT i
82: END SUB
83:
```

1.7 探索

1.7.1 2分木

2分木とは，**1.5**で取り上げたリストの一種で，おのおのの節が2方向のリンクを持つものです．左右のリンクが，指すものを持たない節のことを葉といいます．その節を指すリンクが存在しないものを根といいます．図で表すと図1.7.1のようなもので，○印が節です．節から延びる左右の手がリンクを表します．通常は上から下へリンクするので，矢印はつけません．リンク元の節を親，リンク先の節を子と呼びます．この図を木(tree)というのは，天地を逆にすると植物の木の形に似ているからでしょう．

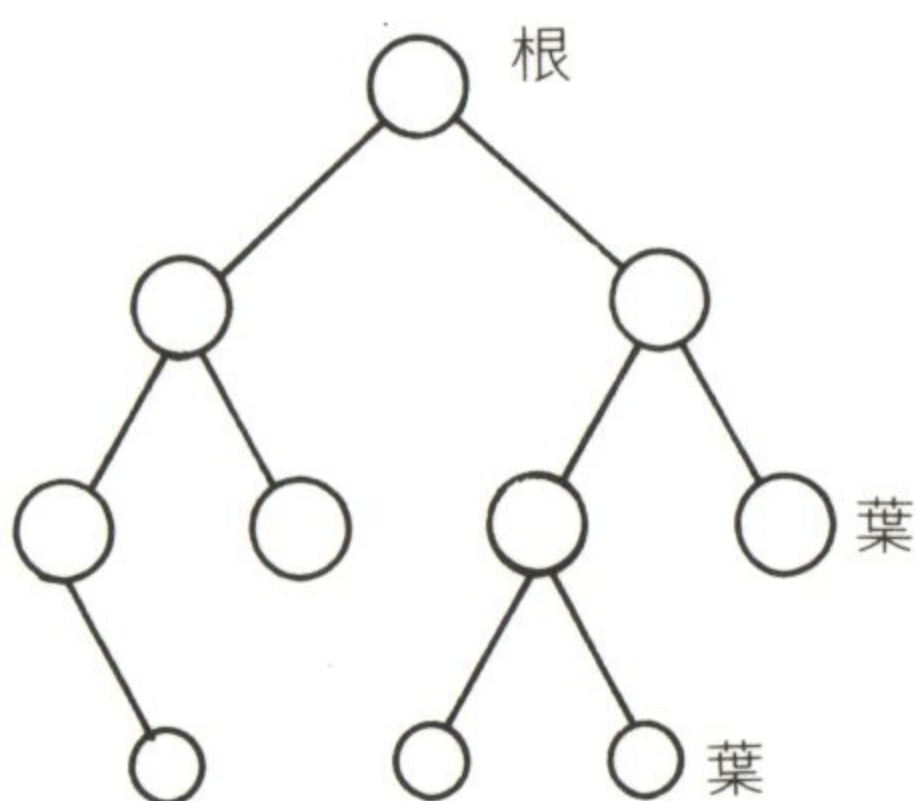
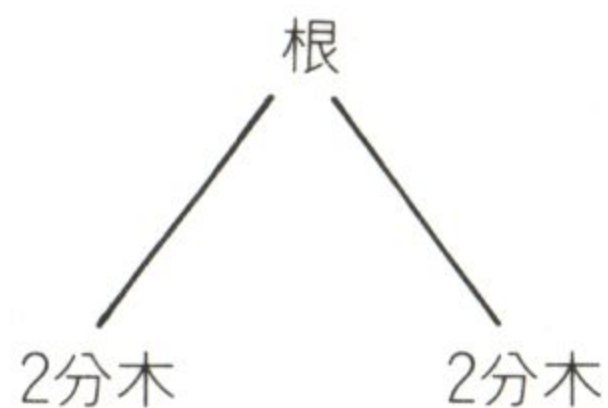
●図1.7.1 2分木

1.7.2 部分木—再帰的構造

図1.7.1で根の左にリンクされた節から先の節は，また木をなしています．したがって２分木は根の左右に２分木をリンクしたものと言えます．

●図1.7.2

根

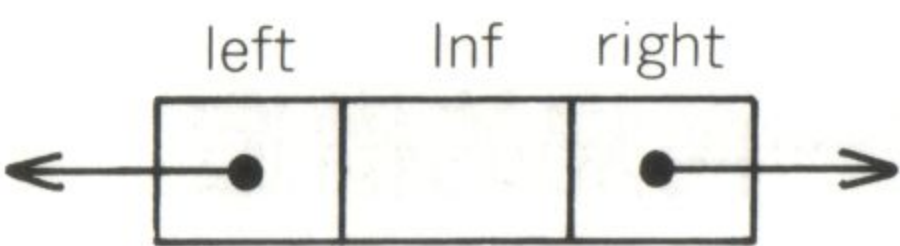
2分木　　2分木

根の先にリンクされた２分木を，部分木と呼びます．部分木は，その根の左右にまた部分木を持ちます．すなわち，２分木は再帰的構造を持ちます．ここで２分木の定義を拡張して，節を１個も持たないものも，節が１個のみ，すなわち根のみのものも，２分木と呼ぶことにします．これは２分木の再帰構造を記述するための形式的な拡張です．

1.7.3 ２分木の実現

1.7.3.1 データ型宣言

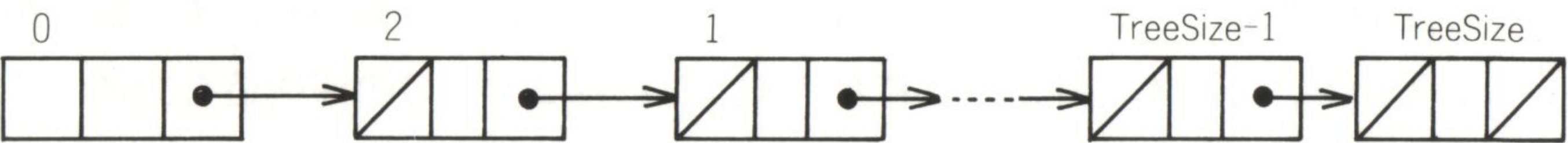
以下の説明のために，２分木を実現するプログラムを示します．QBではポインタが使用できないので，節の配列を用いることにします．

```
TYPE node
        Inf AS STRING *10
        left AS INTEGER
        right AS INTEGER
END TYPE
```

●図1.7.3 データ型nodeの構造

データ型nodeとして，要素Infと左右のリンクを持つ型を定義します(図1.7.3)．最大の木の大きさをTreeSizeとして次のように節を確保します．

```
CONST TreeSize=100
DIM  x(TreeSize) AS node
```

1.5.2の考え方に基づき，はじめにすべての節を自由リストとして構成します．x(0)にはrightとleftの２つのリンクがあるので，rightを自由リストのヘッドを指すものとします．すると，自由リストは次のようになります(図1.7.4参照)．

●図1.7.4 初期化された自由リスト


```
FOR i=1 TO TreeSize
        x(i-1).right=i
NEXT i
x(TreeSize)=Nil
```

Nilは，リスト1.5.1で宣言された定数と同じものとします．新しくデータを挿入する場合には，自由リストからひとつ取り出します．また，木からデータを削除した場合には，削除された節を自由リストに加えます．これらの処理は，**1.5.2**に同じです．

1.7.3.2 挿入

２分木の節に格納される情報について，今までのところ何も定めてありませんでしたが，ここで次のように大小関係を定めます．すなわち，

左の子は親より小さい

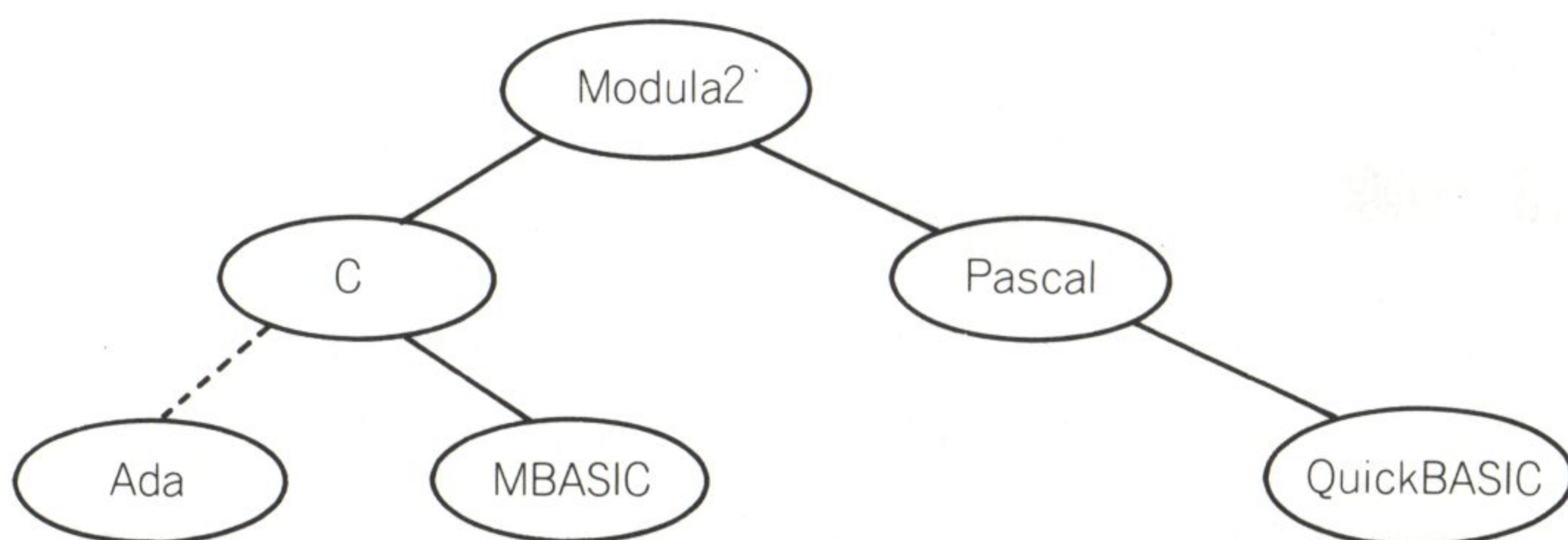
右の子は親に等しいか大きい

この約束で木を構成してゆくと，情報を検索する場合，高速に捜すことが可能になります．

2分木においては，情報の挿入は葉の先になされます．挿入すべき情報と節の大小を比較し，情報が小さければ左の子へ，さもなくば右の子へ挿入します．左または右の子が存在しない場合には，その情報を左または右の子とします．

たとえば，図1.7.5では，5個の情報“Modula 2”，“Pascal”，“C”，“MBASIC”，“QuickBASIC”が，この順に挿入されてできた木に，情報“Ada”を挿入しています．2分木では，一番始めに挿入された情報が根に置かれます．情報“Ada”はまず“Modula 2”と比べられます．小さいので，左の子を調べます．左の子はNilでなく“C”なので，今度は“C”と比べます．“C”より小さいので“C”を含む節の左の子を調べます．左の子は存在しない，すなわち左リンクがNilなので，左リンクに“Ada”を接続します．

●図1.7.5 木への挿入――“Ada”を追加する

Modula2
C
Pascal
Ada
MBASIC
QuickBASIC

この部分をプログラムで書くと次のようになります．p%が調べるリンクを表します．

```
SUB insert(s$,p%)
    IF p%=Nil THEN
        p%=GetNode     '自由リストから領域を確保する
        x(p%).inf=s$ : x(p%).left=Nil : x(p%).right=Nil
    ELSE                    '左または右の子が存在する
        IF s$< x(p%).Inf THEN
            CALL insert(s$,x(p%).left)  '小さければ左の子へ
        ELSE
```

```
          CALL insert(s$,x(p%).right)  'さもなくば右の子へ
        END IF
    END IF
  END SUB
```

飛び込み時のp%の値は，左または右の子へのリンク先の値です．もし，このプロシージャをメイン部から呼ぶ場合には，p%の値は，根を指していなくてはなりません．p%がNilであると子がないので，空きリストから利用可能な節の番号を確保し，それをp%とします．そして，その節にデータs$を書き込み，その左右の子は存在しないのでNilとします．p%へ自由リストからの利用可能な節の番号を書き込んだ時点で，QBのプロシージャの引数がアドレス渡しであることにより，親から子へのリンクも構成されます．

挿入は，単に木の一番下の葉または，左または右の子しか持たない節に付加するので話は簡単です．しかし次に述べる削除は，木の回転と呼ばれる操作が入るので少々面倒です．

1.7.3.3 削除

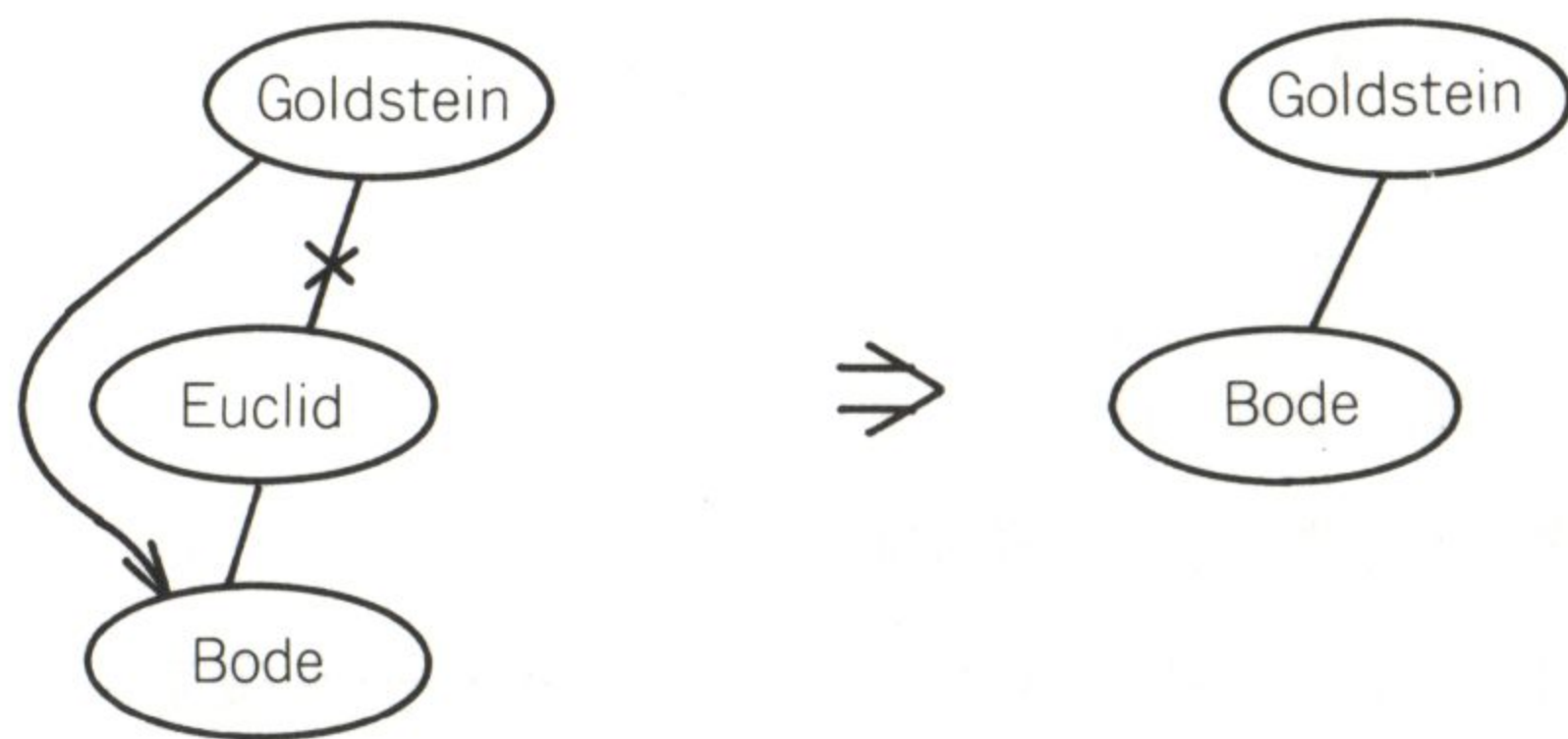
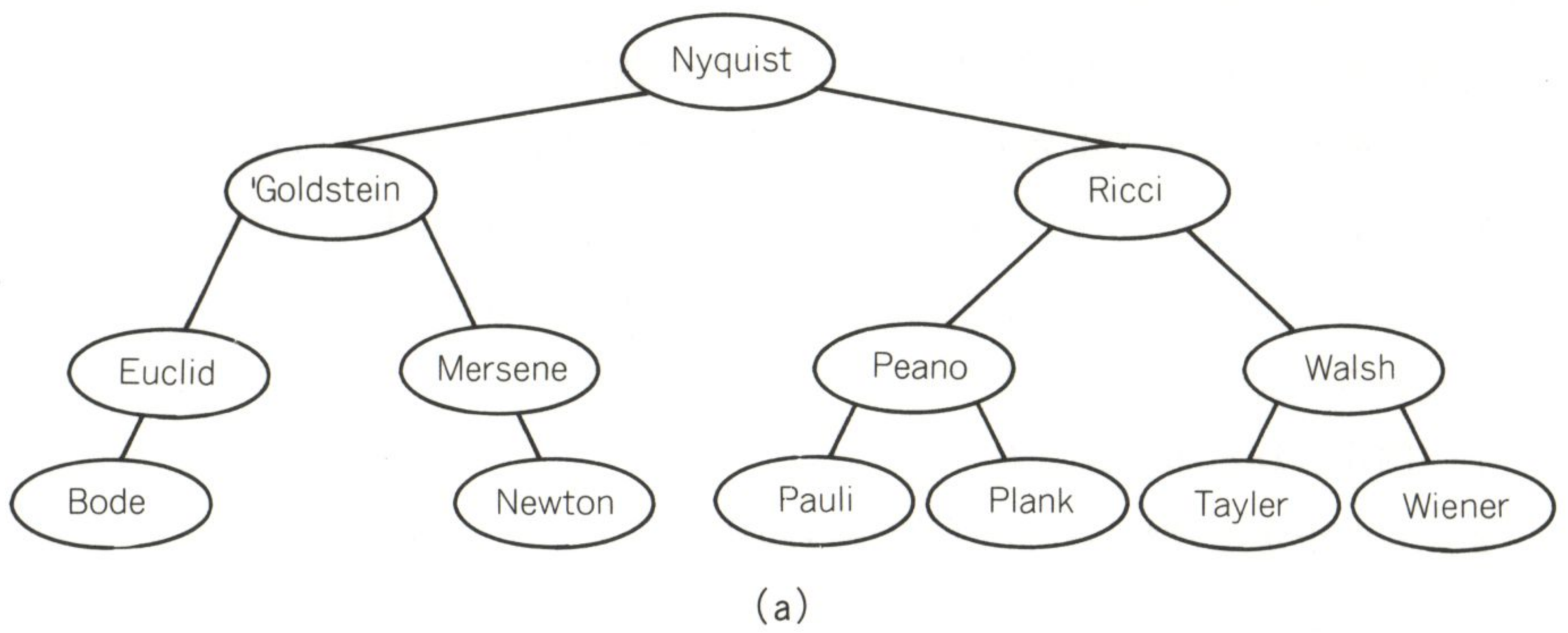
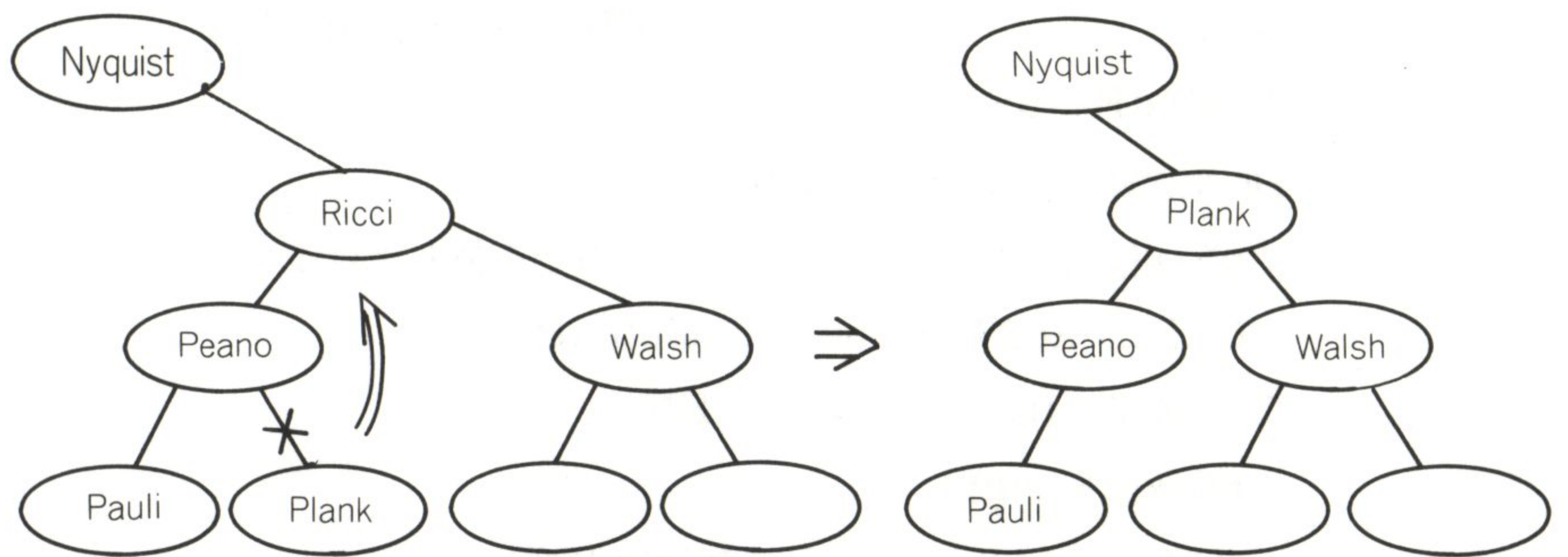
削除の方法を考察するために図1.7.6(a)の木からの削除を調べてみます．節“Bode”，“Newton”，“Pauli”，“Plank”などは葉に相当します．これらの節は，左右のリンクがNilであり，除いても差し支えありません．“Euclid”や“Mersene”のように右または左の子のない節を除いた場合には左または右の子を持ち上げます．“Euclid”を削除する場合，“Goldstein”>“Euclid”>“Bode”より，“Goldstein”>“Bode”の関係は維持されるので，木の性質はこわされません(図1.7.6(b))．

図1.7.6(a)の木で，“Ricci”を除く場合には，回転と呼ばれる操作が必要となります．“Ricci”を除いたあとに“Peano”，“Walsh”のいずれを持ち上げても，2分木を構成できません．たとえば，“Peano”を“Ricci”のあとに置くと，“Plank”は“Peano”の右にこなくてはなりませんが，そこにはすでに“Walsh”があります．

このような場合には，次のように，

削除する節の左右の子が存在する場合には，左の部分木の中の最大の節を，削除された節に置き換える

●図1.7.6

(a)

(b) “Euclid”を削除する

(c) “Ricci”を削除する，回転が生ずる

とします．そうすれば，新しく置き換えられた節の左部分木のすべての節は小さいので2分木の性質は維持されます．

左部分木の最大の節は，各節のリンクを右に順にたどっていって，右リンクがはじめてNilとなった節となります．この例では，“Plank”が最大の節なので，それを“Ricci”のあとに入れます．“Peano”の下にある節が回転するかのように上へ持ち上がるので，この操作を回転と言います(図1.7.6(c))．

まとめると次のようになります．

1．削除される節を見出す
2．右の子がなければ，左の子を持ち上げる
3．左の子がなければ，右の子を持ち上げる
4．左右の子があれば，左の部分子の最大の節を捜し回転する

これをプログラムすると次のようになります．

```
SUB delete(s$,p%)
  IF p%<>Nil THEN
    IF s$<x(p%).Inf THEN
      CALL delete(s$,x(p%).left)   '左の子を捜す
    ELSEIF s$>x(p%).Inf THEN
      CALL delete(s$,x(p%).right) '右の子を捜す
    ELSE                           '削除する節が見出された
      deletedP%=p%
      IF x(p%).right=Nil THEN
        p%=x(p%).left              '左の子を持ち上げる
        CALL  AddToFreeList(deletedP%)
      ELSEIF x(p%).left=Nil THEN
        p%=x(p%).right
        CALL AddToFreeList(deletedP%)
      ELSE
        CALL turn(p%,x(p%).left)   '回転
      END IF
```

```
      END IF
    END IF
END SUB

SUB turn(p%,u%)
    IF x(u%).right<>Nil THEN
          CALL turn(p%,x(u%).right)    '右の子を順に捜す
    ELSE                               '最大の節が見つかった
           x(p%).Inf=x(u%).Inf         '最大の節の内容を書き込む
           deletedU%=u%                '最大の節が捨てられる
           u%=x(u%).left               '左の子を持ち上げる
           CALL AddToFreeList(deletedU%)
    END IF
END SUB
```

挿入のときと同様に引数がアドレス渡しなので，プロシージャ内の引数の書き換えがリンクの接続の変更を実現しています．このプロシージャをメイン部から呼ぶ場合には，引数p%は根を指していなくてはなりません．

1.7.3.4 印刷ートラバーサル

データは木の節に格納されていますので，節の内容を呼び出すことが必要になります．節自体は，配列として実現されていますが，相互関係は左右のリンクで表されます．リンクを順にたどって節の内容を調べることになります．節を順にたどることをトラバーサル(traversal)と呼びます．

トラバーサルの方法を考える場合，木の再帰構造が重要になります．**1.7.2**で述べたように，木は根の下の左右の部分木からなります．したがって，根と左および右の部分木をどのようにたどるかにより，アルゴリズムは変わってきます．基本的には次の3つのトラバーサルの方法が知られています．

●先順トラバーサル(preorder traversal)

1．木が空ならば何もしない

2．さもなくば

根を訪れる

左部分木を訪れる

右部分木を訪れる

●対称トラバーサル(symmetric traversal)

1．木が空ならば何もしない

2．さもなくば

左の部分木を訪れる

根を訪れる

右の部分木を訪れる

●後順トラバーサル(postorder traversal)

1．木が空ならば何もしない

2．さもなくば

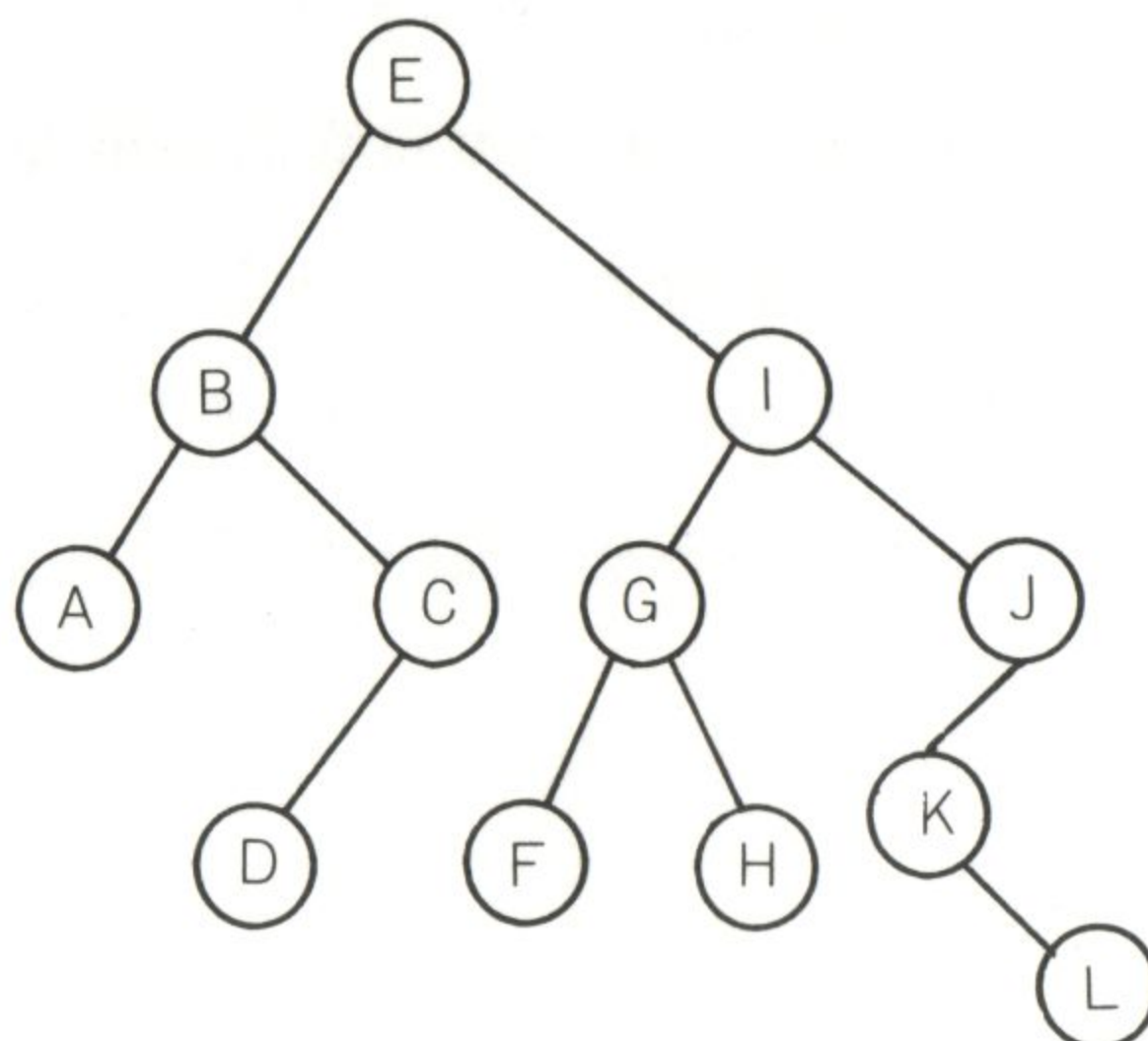
左の部分木を訪れる

右の部分木を訪れる

根を訪れる

上の3つのトラバーサルは,根を訪れる順が最初か中間か最後かが異なります．また，このアルゴリズム自体が再帰になっています．1.が再帰的呼び出しの終了条件になります．

これらの3つのトラバーサルを比較するために，図1.7.7を訪れてみます．2分

●図1.7.7

木の節の情報はアルファベットの大小を満たすように並んでいます.

先順トラバーサル　EBACDIGFHJKL

対称トラバーサル　ABCDEFGHIJKL

後順トラバーサル　ADCBFHGLKJIE

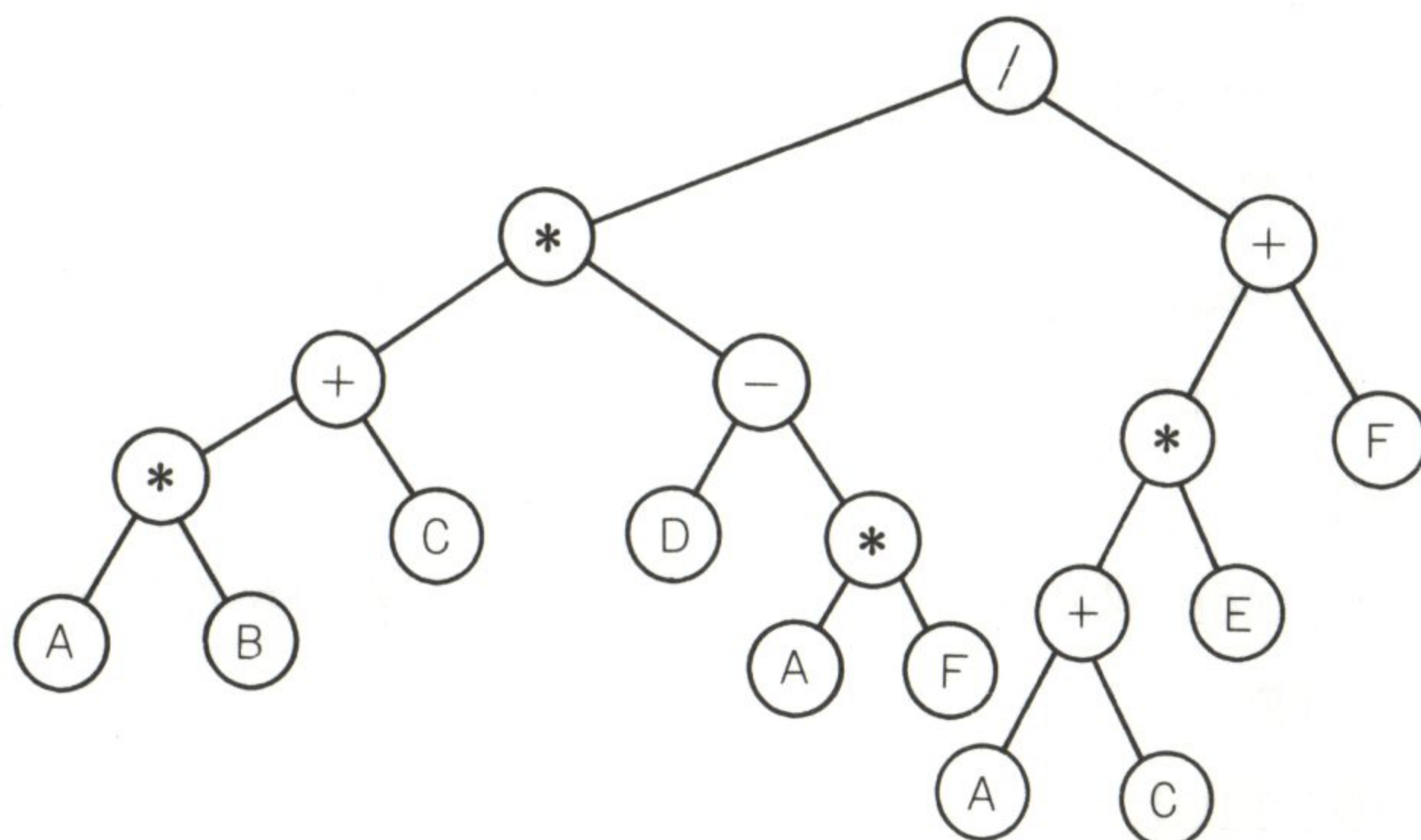
この結果より，左の部分木の情報は根よりも小さく，右部分木のそれは根より大きいとする図1.7.7のような木では,対称トラバーサルによって情報を昇順で取り出すことがわかります.

対称トラバーサルにより，情報を読み出しプリントするプロシージャは次のようになります.

```
SUB PrintTree(p%)
  IF p%<>Nil THEN                         '木が空でなければ
     CALL PrintTree(x(p%).left)           '左の部分木を訪れる
     PRINT  x(p%).Inf                     '根を訪れる
     CALL PrintTree(x(p%).right)          '右の部分木を訪れる
   END IF
END SUB
```

2分木は,代数式の構造を記述するのにもよく使用されます.たとえば,図1.7.8は算術式

(A∗B+C)∗(D−A∗F)/((A+C)∗E+F)

●図1.7.8

を表現しています．この2分木を後順トラバーサルで訪れると次のようになります．

AB＊C＋DAF＊－＊AC＋E＊F＋/

この順序は逆ポーランド記法と呼ばれるものです．

したがって，どのトラバーサルが適しているかは，2分木の性質に依存します．

1.7.3.5 2分木による検索

1.7.3.2挿入の項で述べたように，2分木が"左の子は親より小さい"という約束で節に情報が格納されていると，必要な情報を取り出すのは容易です．たとえば，図1.7.7の2分木で"H"を捜すには4回の比較で済みます．この木は12個の節を持ちますが，"E"で比較し"I"で比較し"G"で比較し，"H"と比較して発見されます．これは2分検索と呼ばれる方法に似ています．

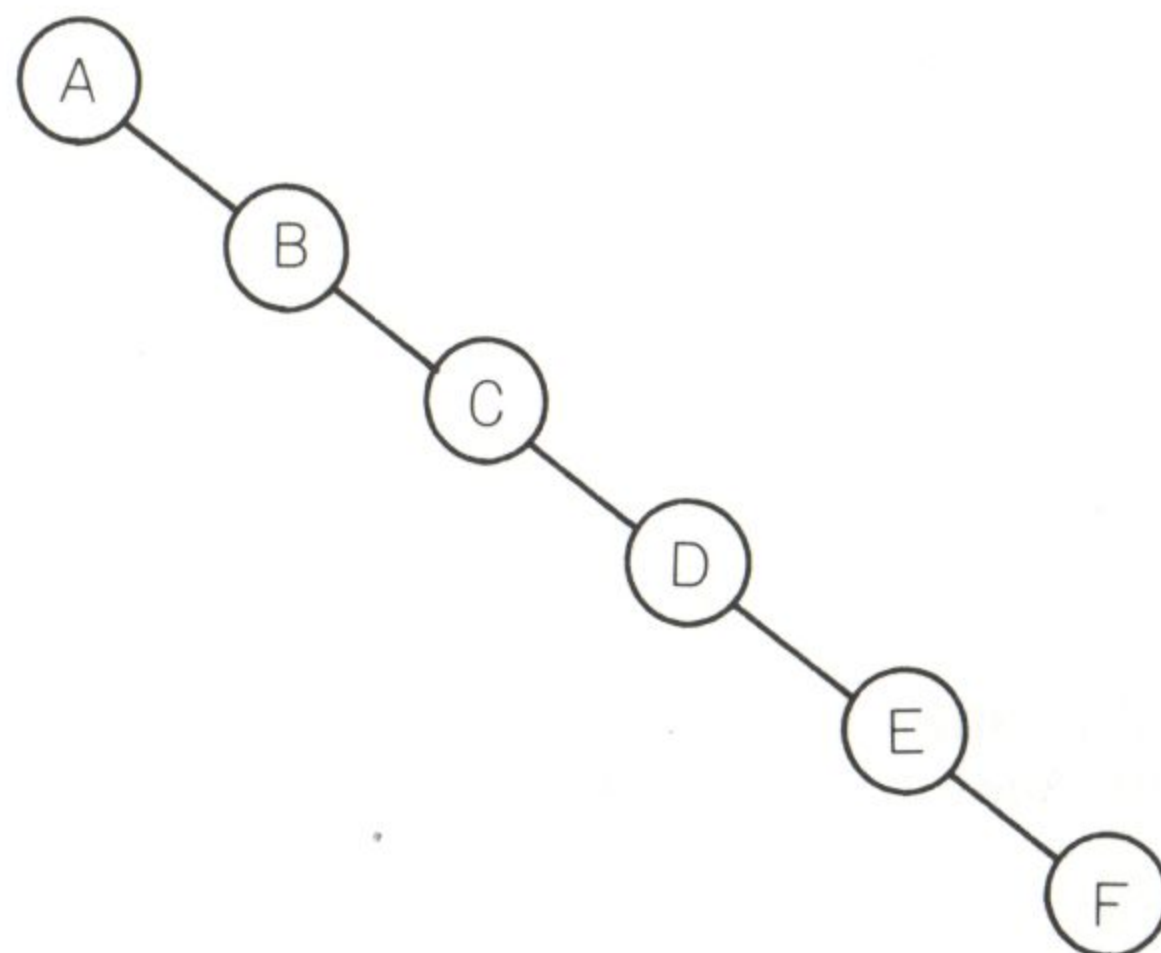
2分木は，もし情報がランダムな順で挿入されると，左右がほぼバランスした形になります．2^n個の情報が完全に左右対称の形に配置されていると，n回の比較で済むことになります．

検索のプログラムは次のようになります．関数の戻り値は，見つかったときは節の添字，見つからない場合にはゼロになります．

```
FUNCTION search(s$,p%)
    IF p%<>Nil THEN
        IF s$<x(p%).Inf THEN
            search=search(s$,x(p%).left)    '左の部分木を捜す
        ELSE IF s$>x(p%).Inf THEN
            search=search(s$,x(p%).right)   '右の部分木を捜す
        ELSE
            search=p%                       '見つかった
        END IF
    ELSE
        search=False                        '見つからなかった
    END IF
END FUNCTION
```

情報があらかじめソートされた状態で与えられると，2分木では，枝は左または右の1方向にのみ作成されます(図1.7.9)．このような木に対する検索は，節の数をNとすると，最も速い場合には1回のみの比較(図では“A”を捜す場合)，最も遅い場合はN回の比較(図では“F”を捜す)，平均すればN/2回となります．Nが大きい場合，これはずいぶん非能率的な検索になるので，図のようなアンバランスな木が作られないように注意する必要があります．

●図1.7.9

A
B
C
D
E
F

2分木がアンバランスにならないように，挿入および削除の時点でバランスを調整する方法としては，ダイナミックバランス木(あるいはAVL-木)が知られています．第2部の単語帳の作成で利用されています．

1.7.3.6 プログラムリスト

全体のプログラムリストを**リスト1.7.1**に示します．メインプログラム(モジュールレベルコード)では，10個のデータを2分木に挿入し，そのあとに“Abel”の有無を調べ，存在しているので削除しています．

プロシージャPrintTreeは，本文で説明した部分以外に，レベルの表示と，左右の子を同時に表示するように機能強化されています．

●リスト 1.7.1　2分木プログラム

```
1: DECLARE FUNCTION Search! (s$, p%)
2: DECLARE SUB InitFreeList ()
3: DECLARE SUB turn (p%, u%)
4: DECLARE SUB delete (s$, p%)
5: DECLARE FUNCTION GetNode! ()
6: DECLARE SUB insert (s$, p%)
7: DECLARE SUB PrintTree (p%, depth!)
8:
```

```
 9: CONST False = 0, True = NOT False, Nil = 0
10: TYPE node
11:   Inf AS STRING * 10
12:   left AS INTEGER
13:   right AS INTEGER
14: END TYPE
15:
16: DIM SHARED TreeSize AS INTEGER
17: TreeSize = 100
18: DIM SHARED x(TreeSize) AS node
19:
20:     CLS
21:     CALL InitFreeList
22:
23:     p% = Nil          'Root Pointer
24:
25:     READ a$
26:     DO WHILE a$ <> "End"
27:         CALL insert(a$, p%)
28:         READ a$
29:     LOOP
30:     CALL PrintTree(p%, 0)
31: INPUT a$
32:     s$ = "Abel      "
33:     IF Search(s$, p%) THEN
34:         CALL delete("Abel      ", p%)
35:     END IF
36:     CALL PrintTree(p%, 0)
37:
38:     DATA "Neumann", "Jaccobi","Abel", "Pascal","Gauss"
39:     DATA "Fuler", "Wiener", "Aitken","Kutta","End"
40:
41: END
42:
43: SUB delete (s$, p%)
44:     IF p% <> Nil THEN
45:         IF s$ < x(p%).Inf THEN
46:             CALL delete(s$, x(p%).left)
47:         ELSE
48:             IF s$ > x(p%).Inf THEN
49:                 CALL delete(s$, x(p%).right)
50:             ELSE
51:                 fl% = True
52:                 IF x(p%).right = Nil THEN
53:                     p% = x(p%).left
54:                 ELSE
55:                     IF x(p%).left = Nil THEN
56:                         p% = x(p%).right
57:                     ELSE
58:                         CALL turn(p%, x(p%).left)
59:                     END IF
60:                 END IF
61:             END IF
62:         END IF
63:     END IF
64:
65: END SUB
66:
67: FUNCTION GetNode
68:
69:      GetNode = x(0).right
```

```
 70:      x(0).right = x(x(0).right).right
 71:
 72: END FUNCTION
 73:
 74: SUB InitFreeList
 75:     FOR i = 1 TO TreeSize
 76:         x(i - 1).right = i
 77:     NEXT i
 78:     x(TreeSize).right = Nil
 79:
 80: END SUB
 81:
 82: SUB insert (s$, p%)
 83:    IF p% = 0 THEN
 84:        p% = GetNode
 85:        IF p% = 0 THEN
 86:            PRINT "No room!!! Insertion '"; s$; "' is aborted."
 87:            EXIT SUB
 88:        END IF
 89:        x(p%).Inf = s$: x(p%).left = Nil: x(p%).right = Nil
 90:    ELSE
 91:        IF s$ < x(p%).Inf THEN
 92:            CALL insert(s$, x(p%).left)
 93:        ELSE
 94:            CALL insert(s$, x(p%).right)
 95:        END IF
 96:    END IF
 97: END SUB
 98:
 99: SUB PrintTree (p%, depth)
100:    IF p% <> 0 THEN
101:        CALL PrintTree(x(p%).left, depth + 1)
102:        l$ = x(x(p%).left).Inf: r$ = x(x(p%).right).Inf
103:        PRINT USING "###: @  @ @ "; depth; l$; x(p%).Inf; r$
104:        CALL PrintTree(x(p%).right, depth + 1)
105:    END IF
106: END SUB
107:
108: FUNCTION Search (s$, p%)
109:    IF p% <> Nil THEN
110:        IF s$ < x(p%).Inf THEN
111:            Search = Search(s$, x(p%).left)      'search left tree
112:        ELSEIF s$ > x(p%).Inf THEN
113:            Search = Search(s$, x(p%).right)     'search right tree
114:        ELSE
115:            Search = p%                          'Found
116:        END IF
117:     ELSE
118:        Search = False                           'Not found
119:     END IF
120:
121: END FUNCTION
122:
123: SUB turn (p%, u%)
124:  IF x(u%).right <> Nil THEN
125:     CALL turn(p%, x(u%).right)
126:  ELSE
127:     x(p%).Inf = x(u%).Inf
128:     u% = x(u%).left
129:  END IF
130: END SUB
```

第2部
Quick BASIC
プログラム研究

Quick BASIC

2.1 Quick BASIC ソースリスト清書プログラム

2.1.1 概　要

本章では，応用例として，テキストモードでセーブされたQB用のソースリストをプリンタに印刷するプログラムを紹介します．プログラム名をFINELST.BAS (Fine Listing)とします．

プログラムの概要は次のとおりです．

- 1行に印刷される文字数を増やすために英文字はエリートサイズ（1インチ12文字)とし，A4用紙あるいは10″の連続紙の場合96文字で印字する．
- キーワードは強調文字で印字し，読みやすくする．
- 使いやすさを考えて，印刷するファイルは，画面に一覧されたファイルリストから選択可能とする．

ここで使用されるプログラム上の技法の主なものは次のとおりです．

- インタラプト命令を使用して，DOSのシステムコールを実行する．
- プログラム中からのプリンタの制御についての実例を示す．
- テキスト中から，デリミタで区切られた単語を摘出する．
- 文字コード判定のためのいくつかの関数を作成する．

2.1.2 基本方針

清書プログラムは文字どおり，ソースリストを美しく印刷するプログラムです．“美しく”とは，人によっても価値基準は異なるでしょうが，ここでは実用的な観点から次のような基本仕様を定めます．

（1） 出力用紙に印刷されたソースリストの各行に通し番号を付し，かつページをつけて，散逸を防ぐ．

（2） ページごとにファイル名とファイル作成日を印字する．これにより，ファイルのバージョンの相違が区別できる．かつ，印刷した日付も付す．

（3） キーワードは強調文字で印字することにし，リストに変化をつける．これによって可読性が向上する．

（4） 1行の文字数が多い方が印字は楽であるが，文字が小さいと読みにくくなるので，エリートサイズ（1インチ，12文字）を使用する．これにより，行ごとに行番号を付しても，エディタの画面サイズ80桁はカバーできる．

以上のような要求を満たすプログラムを開発しますが，具体的には以下のようなことを定めておく必要があります．

（5） 印字すべきファイルは，テキストモードでファイルされていること．

（6） プリンタはPR-201系とする（プリンタのコントロール部については後述するので，ほかのプリンタを使用する場合はそれを変更してください）．

（7） 1行が長い場合には，自動的に複数行に分けて印字する．その際，“語”が分割されないようにする．

プログラムのコーディングに際しては，次のような方針をたてます．

（8） 冗長になっても，可読性がよくなることを心がける．

（9） QBで用意された使いやすい制御構造を積極的に使用してみる．従来のBASICの命令はできるだけさける（たとえば，1行のIF THEN ELSEを使用しないとか，DEF FNの代わりにFUNCTIONとするなど）．

（10） プロシージャ，関数を多用し，かつ，それらの使用によりプログラムが読

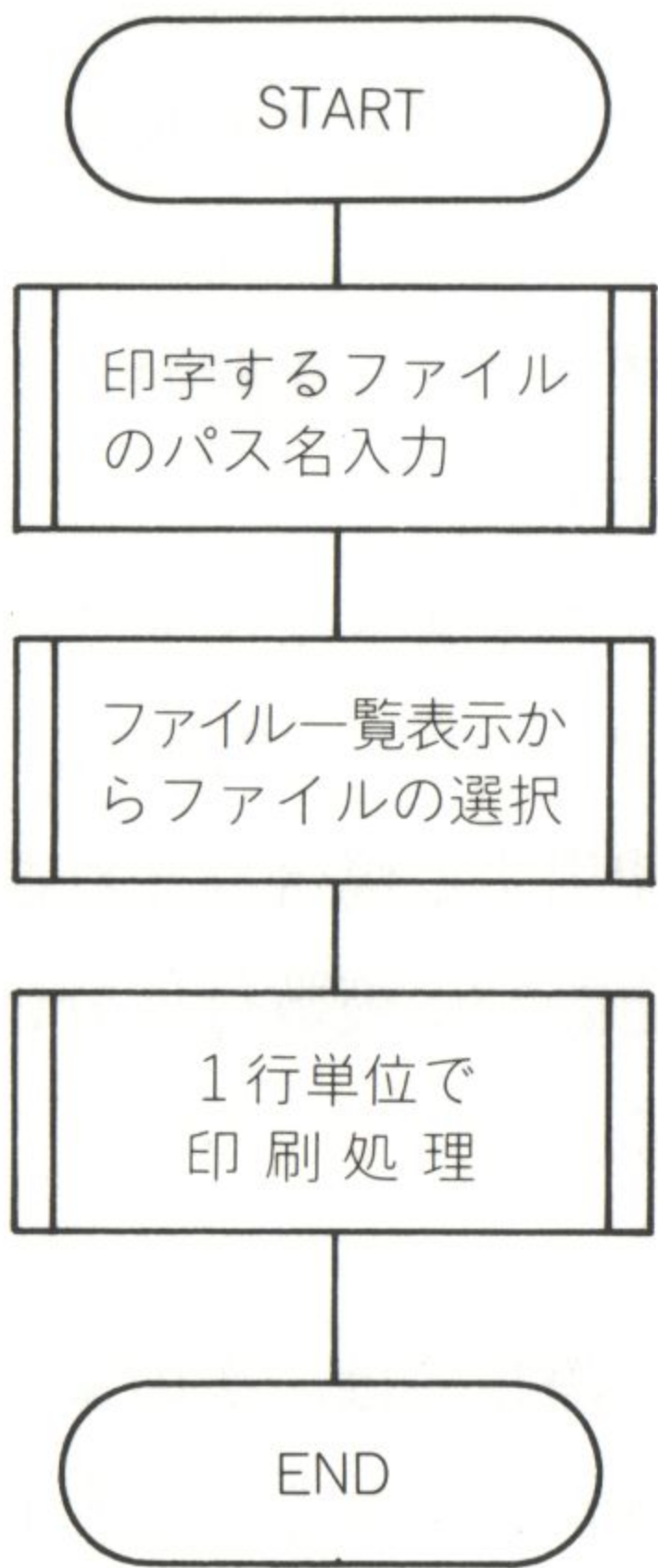
みやすくなるようにする．

以上の方針で作業を進めます．

2.1.3 プログラム概要

このプログラムは，印字すべきテキストファイルの選択を，画面に一覧されたファイルリストから行ったのち，テキストファイルを1行ずつフロッピィディスクから読み込んで，可読性のよいリストに整形したのちプリンタに出力していきます．

図2.1.1にプログラム全体のゼネラルフローを示します．また，リスト2.1.1にプログラム全体のリストを示します．本書のプログラムのリストはすべてこの清書プログラムを使っています．

●図2.1.1 清書プログラムゼネラルフローチャート

●リスト 2.1.1

```
 1: DECLARE SUB GetDir (inpfn$, inpdev$, optdev$, title$)
 2: DECLARE SUB DirSub (sumfile!, file$(), path$)
 3: DECLARE FUNCTION FileName$ (s$)
 4: DECLARE FUNCTION IndentN! (g$)
 5: DECLARE FUNCTION PutZero$ (m$)
 6: DECLARE FUNCTION IsDigit! (c$)
 7: DECLARE FUNCTION IsVar! (b$)
 8: DECLARE FUNCTION IsAlpha! (a$)
 9: DECLARE FUNCTION DispSecd% (t%)
10: DECLARE FUNCTION DispHour% (t%)
11: DECLARE FUNCTION DispMint% (t%)
12: DECLARE FUNCTION DispMonth% (d%)
13: DECLARE FUNCTION DispDay% (d%)
14: DECLARE FUNCTION DispYear% (d%)
15: DECLARE SUB ProcTab (p$)
16: DECLARE FUNCTION IsShiftJis (ch$)
17: DECLARE SUB LnPrint (oneline$, a$, keyw$(), keysize)
18: DECLARE FUNCTION SearchKey! (keyw$(), a$, keysize!)
19: DECLARE SUB SetKey (keyw$(), keysize)
20: DECLARE SUB FFeed (p!, l!, inpfn$, title$)
21: DECLARE SUB TitlPr (p!, inpfn$, title$)
22: DECLARE SUB PR (clnum, rwnum, nnum, inpfn$, inpdev$, optdev$, keyw$(), keysize, title$)
23: DECLARE FUNCTION NumArea! (n!, tn!)
24: DECLARE SUB SelFile (sumfile!, file$(), escflg, filenum)
25: DECLARE FUNCTION RowPos! (n!)
26: DECLARE FUNCTION ColPos! (n!)
27: DECLARE SUB DispFile (sumfile!, file$())
28:
29: '$INCLUDE: 'a:¥qb45¥INCLUDE¥QB.BI'
30:
31: DIM SHARED dstmp(1 TO 200) AS STRING * 2
32: DIM SHARED tstmp(1 TO 200) AS STRING * 2
33: DIM SHARED keyw$(50)
34:
35: CONST LINETOP = 4, LINEEND = 24, FALSE = 0, TRUE = NOT FALSE
36:
37: ON ERROR GOTO ErrExit
38:
39: clnum = 96
40: rwnum = 58
41: nnum = 5
42:
43: WIDTH "LPT1:", 255
44:
45: CALL GetDir(inpfn$, inpdev$, optdev$, title$)
46:
47: CALL SetKey(keyw$(), keysize)
48:
49: CALL PR(clnum, rwnum, nnum, inpfn$, inpdev$, optdev$, keyw$(), keysize, title$)
50:
51: CLS
52:
53: END
54:
55: DATA AS, CALL, CASE, CLOSE, COMMON, CONST, DATA, DECLARE, DEF, DEFDBL
56: DATA DEFINT, DEFLNG, DEFSNG, DEFSTR, DIM, DO, ELSE, ELSEIF, END, ENDIF
57: DATA EXIT, FOR, FUNCTION, GOSUB, GOTO, IF, LOOP, NEXT, OPEN, REM
58: DATA RESUME, RETURN, SELECT, STATIC, STOP, SUB, THEN, TO, TYPE
59: DATA UNTIL ,WEND, WHILE, z
60:
61: ErrExit:
62:
63:     PRINT "エラーが発生しました. "
64:     PRINT "エラー番号 = "; ERR
65:
66:     SELECT CASE ERR
67:         CASE 71
```

```
 68:            PRINT "指定したドライブにファイルがありません"
 69:            PRINT "ファイルの入ったフロッピィディスクをセットして"
 70:            PRINT "なにかキーを押してください."
 71:            DO
 72:            LOOP UNTIL (INKEY$ <> "")
 73:        CASE ELSE
 74:            CLOSE
 75:            END
 76:
 77:    END SELECT
 78:
 79:
 80: RESUME
 81:
 82: FUNCTION ColPos (n)
 83:
 84:    ColPos = (1 + 16 * (n - 1)) MOD 80
 85:
 86: END FUNCTION
 87:
 88: SUB DirSub (sumfile, file$(), path$)
 89:
 90: DIM R AS RegTypeX
 91: DIM dta AS STRING * 129
 92: DIM fld AS STRING * 128
 93: DIM flag(1 TO 200) AS STRING * 1
 94:
 95: sumfile = 0
 96: dta = SPACE$(129)
 97: fld = path$ + SPACE$(128 - LEN(path$))
 98: MID$(fld, LEN(path$) + 1) = CHR$(0)
 99: R.ax = &H1A00
100: R.ds = VARSEG(dta)
101: R.dx = VARPTR(dta)
102: R.es = -1
103: CALL INTERRUPTX(&H21, R, R)
104:
105: R.ax = &H4E00
106: R.ds = VARSEG(fld)
107: R.dx = VARPTR(fld)
108: R.es = -1
109: R.cx = &H0
110: CALL INTERRUPTX(&H21, R, R)
111: IF (R.ax = &H12 OR R.ax = &H2 OR R.ax = &H3) THEN GOTO skip
112:
113: DO
114:    sumfile = sumfile + 1
115:    flag(sumfile) = MID$(dta, &H16, 1)
116:    file$(sumfile) = MID$(dta, &H1F, 12)
117:    tstmp(sumfile) = MID$(dta, &H17, 2)
118:    dstmp(sumfile) = MID$(dta, &H19, 2)
119:
120:    MID$(dta, &H1F, 12) = SPACE$(12)
121:    MID$(dta, &H17, 2) = SPACE$(2)
122:    MID$(dta, &H19, 2) = SPACE$(2)
123:    R.ax = &H4F00
124:    R.dx = VARPTR(fld)
125:    R.ds = VARSEG(fld)
126:    R.es = -1
127:    R.cx = &H0
128:    CALL INTERRUPTX(&H21, R, R)
129: LOOP UNTIL (R.ax = &H12) OR (sumfile >= 200)
130:
131: skip:
132:
133: IF (R.ax = &H2 OR R.ax = &H3) THEN
134:     PRINT "指定したドライブ(またはパス)に"
135:     PRINT "フロッピィディスクがないか,"
136:     PRINT "ファイルが存在しません."
```

```
137:        END
138: END IF
139:
140:
141: END SUB
142:
143: FUNCTION DispDay% (d%)
144:
145:     DispDay% = d% AND &H1F
146:
147: END FUNCTION
148:
149: SUB DispFile (sumfile, file$())
150:
151:     VIEW PRINT LINETOP TO LINEEND
152:
153:     row = 3
154:     col = 1
155:
156:     i = 1
157:     LOCATE RowPos(i), ColPos(i)
158:     COLOR 15
159:     PRINT USING "&              &"; file$(i)
160:
161:     COLOR 7
162:     FOR i = 2 TO sumfile
163:             LOCATE RowPos(i), ColPos(i)
164:             PRINT USING "&              &"; file$(i)
165:     NEXT i
166:
167: END SUB
168:
169: FUNCTION DispHour% (t%)
170:
171:     a% = t% AND &HF800
172:
173:     IF a% < 0 THEN
174:         y = a% + 65536
175:         DispHour% = y / &H800
176:     ELSE
177:         DispHour% = a% / &H800
178:     END IF
179:
180: END FUNCTION
181:
182: FUNCTION DispMint% (t%)
183:
184:     a% = t% AND &H7E0
185:
186:     DispMint% = a% / &H20
187:
188: END FUNCTION
189:
190: FUNCTION DispMonth% (d%)
191:
192:     a% = d% AND &H1E0
193:
194:     DispMonth = a% / &H20
195:
196: END FUNCTION
197:
198: FUNCTION DispSecd% (t%)
199:
200:     DispSecd% = t% AND &H1F
201:
202: END FUNCTION
203:
204: FUNCTION DispYear% (d%)
205:
```

```
206:     a% = d% AND &HFE00
207:
208:     DispYear% = a% / &H200
209:
210: END FUNCTION
211:
212: SUB FFeed (p, l, inpfn$, title$)
213:
214:     PRINT #2, " "
215:     PRINT #2, CHR$(12)
216:     p = p + 1
217:     l = 0
218:
219:     CALL TitlPr(p, inpfn$, title$)
220:
221: END SUB
222:
223: FUNCTION FileName$ (s$)
224:
225:     f$ = RTRIM$(s$)
226:
227:     IF ASC(RIGHT$(f$, 1)) = 0 THEN f$ = LEFT$(f$, LEN(f$) - 1)
228:
229:     FileName$ = f$
230:
231: END FUNCTION
232:
233: SUB GetDir (inpfn$, inpdev$, optdev$, title$)
234:
235: DIM file$(1 TO 200)
236:
237: DO
238:     CLS
239:
240:     PRINT "ドライブ名（サブディレクトリの場合はパス名を含む）"
241:     INPUT "を入力してください．例）B:とかB:¥PROGなど"; fdpath$
242:     IF fdpath$ = "" THEN
243:         path$ = "*.*"
244:     ELSE
245:         fdpath$ = fdpath$ + "¥"
246:         path$ = fdpath$ + "*.*"
247:     END IF
248:
249:     CALL DirSub(sumfile, file$(), path$)
250:
251:     escflg = FALSE
252:
253:     CALL DispFile(sumfile, file$())
254:
255:     CALL SelFile(sumfile, file$(), escflg, filenum)
256:
257: LOOP WHILE escflg
258:
259:
260: inpfn$ = FileName$(file$(filenum))
261:
262: inpdev$ = fdpath$ + inpfn$
263:
264: optdev$ = "LPT1:"
265:
266: ds% = CVI(dstmp(filenum))
267: ts% = CVI(tstmp(filenum))
268:
269: lastdate$ = PutZero$(RIGHT$(STR$(1980 + DispYear%(ds%)), 4) + "-" +
     RIGHT$(STR$(DispMonth%(ds%)), 2) + "-" + RIGHT$(STR$(DispDay%(ds%)), 2))
270: lasttime$ = PutZero$(RIGHT$(STR$(DispHour%(ts%)), 2) + ":" + RIGHT$(STR$(DispMint%(ts%))
     2) + ":" + RIGHT$(STR$(DispSecd%(ts%)), 2))
271: nowdate$ = DATE$
272: nowtime$ = TIME$
```

```
273:
274: title$ = inpfn$ + "    " + lastdate$ + "    " + lasttime$ + "      " + nowdate$ + "   " +
     nowtime$
275:
276: END SUB
277:
278: FUNCTION IndentN (g$)
279:
280:     n = 0
281:     DO
282:         n = n + 1
283:         s$ = MID$(g$, n, 1)
284:     LOOP UNTIL s$ <> " "
285:     n = n - 1
286:
287:     IndentN = n
288:
289: END FUNCTION
290:
291: FUNCTION IsAlpha (a$)
292:
293:     IF ((a$ >= "A" AND a$ <= "Z") OR (a$ >= "a" AND a$ <= "z")) THEN
294:         IsAlpha = TRUE
295:     ELSE
296:         IsAlpha = FALSE
297:     END IF
298:
299: END FUNCTION
300:
301: FUNCTION IsDigit (c$)
302:
303:     IF (c$ >= "0" AND c$ <= "9") THEN IsDigit = TRUE ELSE IsDigit = FALSE
304:
305: END FUNCTION
306:
307: FUNCTION IsShiftJis (ch$)
308:
309:     IF ch$ = "" THEN
310:        c = 0
311:     ELSE
312:        c = ASC(ch$)
313:     END IF
314:
315:     IF (((128 < c) AND (c < 160)) OR ((224 <= c) AND (c <= 254))) THEN
316:        IsShiftJis = TRUE
317:     ELSE
318:        IsShiftJis = FALSE
319:     END IF
320:
321: END FUNCTION
322:
323: FUNCTION IsVar (b$)
324:
325:     IF ((b$ >= "A" AND b$ <= "Z") OR (b$ >= "a" AND b$ <= "z") OR (b$ >= "0" AND b$ <= _
     "9") OR b$ = "." OR b$ = "!" OR b$ = "#" OR b$ = "$" OR b$ = "%" OR b$ = "&") THEN
326:         IsVar = TRUE
327:     ELSE
328:         IsVar = FALSE
329:     END IF
330:
331: END FUNCTION
332:
333: SUB LnPrint (oneline$, a$, keyw$(), keysize)
334:
335:     rf = FALSE
336:     qf = FALSE
337:     kflag = FALSE
338:
339:     llen = LEN(oneline$)
```

```
340:     j = 0
341:     DO WHILE j < llen
342:
343:         IF (NOT kflag) THEN
344:             PRINT #2, CHR$(27) + "E";
345:         END IF
346:
347:         j = j + 1
348:         ch$ = MID$(oneline$, j, 1)
349:         kflag = IsShiftJis(ch$)
350:
351:         IF IsAlpha(ch$) THEN
352:             k = 0
353:             a$ = ""
354:             DO
355:                 k = k + 1
356:                 a$ = a$ + ch$
357:                 j = j + 1
358:                 ch$ = MID$(oneline$, j, 1)
359:                 kflag = IsShiftJis(ch$)
360:             LOOP UNTIL (NOT (IsVar(ch$) OR IsDigit(ch$)) OR j > llen)
361:
362:             j = j - 1
363:
364:             IF rf OR qf THEN
365:                 PRINT #2, a$;
366:             ELSE
367:                 IF keyw$(SearchKey(keyw$(), a$, keysize)) = a$ THEN
368:                     PRINT #2, CHR$(27) + "!"; a$; CHR$(27) + CHR$(34);
369:                 ELSE
370:                     PRINT #2, a$;
371:                 END IF
372:             END IF
373:         ELSE
374:             IF ((ch$ = "'") AND (NOT qf)) THEN rf = NOT rf
375:             IF ((ch$ = CHR$(34)) AND (NOT rf)) THEN qf = NOT qf
376:             PRINT #2, ch$;
377:         END IF
378:     LOOP
379:
380:     PRINT #2, " "
381:
382: END SUB
383:
384: FUNCTION NumArea (n, tn)
385:
386:     IF (n >= 1 AND n <= tn) THEN NumArea = TRUE ELSE NumArea = FALSE
387:
388: END FUNCTION
389:
390: SUB PR (clnum, rwnum, nnum, inpfn$, inpdev$, optdev$, keyw$(), keysize, title$)
391:
392:     DIM prfile$(100)
393:
394:     OPEN inpdev$ FOR INPUT AS #1
395:     OPEN optdev$ FOR OUTPUT AS #2
396:
397:     PRINT #2, CHR$(27) + "E"
398:
399:     lnum = 0
400:
401:     page = 1
402:     lincnt = 0
403:
404:     CALL TitlPr(page, inpfn$, title$)
405:
406:     DO UNTIL EOF(1)
407:
408:         i = 1
```

```
409:        lnum = lnum + 1
410:
411:        LINE INPUT #1, prfile$(i)
412:
413:        CALL ProcTab(prfile$(i))
414:
415:        indnt = IndentN(prfile$(i))
416:
417:        prfile$(i) = RIGHT$(STRING$(nnum - 2, " ") + STR$(lnum) + ": ", nnum) + prfile$(i)
418:
419:        DO WHILE LEN(prfile$(i)) > clnum
420:            IF MID$(prfile$(i), clnum, 1) <> " " THEN
421:                k = 1
422:                DO
423:                    IF (MID$(prfile$(i), clnum - k, 1) = " " OR MID$(prfile$(i), clnum -
                    k, 1) = "") THEN EXIT DO
424:                    k = k + 1
425:                LOOP
426:                prfile$(i + 1) = STRING$(indnt, " ") + STRING$(nnum, " ") + _
                MID$(prfile$(i), clnum - k + 1)
427:                prfile$(i) = LEFT$(prfile$(i), clnum - k) + " _"
428:            ELSE
429:                prfile$(i + 1) = STRING$(indnt, " ") + STRING$(nnum, " ") + _
                MID$(prfile$(i), clnum + 1)
430:                prfile$(i) = LEFT$(prfile$(i), clnum) + " _"
431:            END IF
432:            indnt = IndentN(prfile$(i + 1)) - nnum
433:            i = i + 1
434:        LOOP
435:
436:        FOR j = 1 TO i
437:            CALL LnPrint(prfile$(j), a$, keyw$(), keysize)
438:            lincnt = lincnt + 1
439:            IF (lincnt = rwnum - 2) THEN CALL FFeed(page, lincnt, inpfn$, title$)
440:        NEXT j
441:
442:    LOOP
443:
444:    CLOSE #1, #2
445:
446: END SUB
447:
448: SUB ProcTab (p$)
449:
450:    CONST tabval = 4
451:
452:    llen = LEN(p$)
453:    j = 1
454:
455:    DO WHILE j <= llen
456:        ch$ = MID$(p$, j, 1)
457:        IF ch$ = CHR$(9) THEN
458:            p$ = LEFT$(p$, j - 1) + STRING$(tabval, " ") + MID$(p$, j + 1)
459:            llen = LEN(p$)
460:        END IF
461:        j = j + 1
462:    LOOP
463:
464: END SUB
465:
466: FUNCTION PutZero$ (m$)
467:
468:    zeropos = INSTR(m$, " ")
469:    DO WHILE (zeropos)
470:        MID$(m$, zeropos) = "0"
471:        zeropos = INSTR(m$, " ")
472:    LOOP
473:
474:    PutZero$ = m$
```

```
475:
476: END FUNCTION
477:
478: FUNCTION RowPos (n)
479:
480:     RowPos = INT((1 + 16 * (n - 1)) / 80) + LINETOP
481:
482: END FUNCTION
483:
484: FUNCTION SearchKey (keyw$(), a$, keysize)
485:
486:     h = 0
487:     t = keysize
488:
489:     DO
490:        i = (h + t) ¥ 2
491:        IF keyw$(i) <= a$ THEN h = i + 1
492:        IF keyw$(i) >= a$ THEN t = i - 1
493:     LOOP UNTIL h > t
494:
495:     SearchKey = i
496:
497: END FUNCTION
498:
499: SUB SelFile (sumfile, file$(), escflg, filenum)
500:
501:     filenum = 1
502:
503:     DO
504:         DO
505:             a$ = INKEY$
506:         LOOP WHILE a$ = ""
507:
508:         DO WHILE (a$ = CHR$(0, &H48) OR a$ = CHR$(0, &H50) OR a$ = CHR$(0, &H4B) OR a$ =
            CHR$(0, &H4D))
509:
510:             SELECT CASE a$
511:                 CASE CHR$(0, &H48)
512:                     nextfnum = filenum - 5
513:                 CASE CHR$(0, &H50)
514:                     nextfnum = filenum + 5
515:                 CASE CHR$(0, &H4B)
516:                     nextfnum = filenum - 1
517:                 CASE ELSE
518:                     nextfnum = filenum + 1
519:             END SELECT
520:
521:             IF NumArea(nextfnum, sumfile) THEN
522:                 COLOR 7
523:                 LOCATE RowPos(filenum), ColPos(filenum)
524:                 PRINT USING "&              &"; file$(filenum)
525:
526:                 COLOR 15
527:                 LOCATE RowPos(nextfnum), ColPos(nextfnum)
528:                 PRINT USING "&              &"; file$(nextfnum)
529:
530:                 filenum = nextfnum
531:
532:             END IF
533:
534:             DO
535:                 a$ = INKEY$
536:             LOOP WHILE a$ = ""
537:         LOOP
538:
539:     LOOP UNTIL (a$ = CHR$(13) OR a$ = CHR$(&H1B))
540:
541:     IF a$ = CHR$(&H1B) THEN escflg = TRUE
542:
```

```
543:      COLOR 7
544:      VIEW PRINT 1 TO 25
545:
546: END SUB
547:
548: SUB SetKey (keyw$(), n)
549:
550:      i = 0
551:
552:      DO
553:          READ keyw$(i)
554:          IF keyw$(i) = "z" THEN EXIT DO
555:          i = i + 1
556:      LOOP
557:
558:      n = i - 1
559:
560: END SUB
561:
562: SUB TitlPr (p, inpfn$, title$)
563:
564:      hdline$ = title$ + "        " + "page " + STR$(p)
565:
566:      PRINT #2, hdline$
567:      PRINT #2, " "
568:
569: END SUB
570:
```

2.1.4 主要なプロシージャ・関数についての説明

2.1.4.1 DOSのシステムコールの実行(SUBプロシージャ DirSub)

このプログラムでは，印刷するファイルを選択する場合に，画面に一覧されたファイルリストから，カーソルで選択可能な仕様にしています．また，ファイルのバージョンの違いを区別できるように，印字されたリストにはファイル名とともにファイル作成日も付すことにしています．これらは，MS-DOSにおけるDIRコマンドで得られる情報の一部と同じですが，BASICの命令の中にはそれに相当する命令はありません．そこで，このプログラムでは，インタラプト命令を使用し，DOSのシステムコールを利用して，フロッピィディスク内のファイルのファイル名とファイル作成日を得ています．ここではその方法について簡単に説明します．

QBの中からMS-DOSの提供するサービスを利用するためには，ソフトウェア割り込みの命令である，CALL INTERRUPTX命令を使用します．

この関数の書式は，

```
CALL INTERRUPTX(interruptnum,inregs,outreg)
```

です．ここで，interruptnumはDOSの割り込み番号を示す値です．inregsは割り込み実行中のレジスタの値を示す変数で，outregは割り込み後のレジスタの値を示す変数です．この2つの変数はRegTypeX型として宣言しておきます．

さて，MS-DOSのシステムコールの大部分は，割り込み番号&H21で実行されます．実行したい処理のファンクション番号とそのほか必要な情報は，決められたレジスタにセットしておく必要があります．

では，本プログラムにおける場合について具体的に説明しましょう(SUBプロシージャDirSubのプログラムリストを参照してください)．

DOSのシステムコールに関するいくつかの変数を次に示すように宣言しています．

```
DIM R AS RegTypeX
DIM dta AS STRING * 129
DIM fld AS STRING * 128
DIM flag(1 TO 200) AS STRING * 1
DIM file$(1 TO 200)
DIM dstmp(1 TO 200) AS STRING * 2
DIM tstmp(1 TO 200) AS STRING * 2
```

宣言文の先頭で，RegTypeX型の変数Rを宣言しています．

DTA(バッファ)は129文字の固定長文字列型として宣言されています．固定長文字列変数fldは，ディレクトリを指示するパス名を格納するバッファです．

システムコールを使って得られる情報のうち，ファイル名，ファイル作成日および時間は，文字型配列file$，dstmp，tstmpにそれぞれ格納されます．

さて，このプログラムで行うのはDOSのシステムコールを利用した，指定されたディレクトリにおけるファイルの検索です．これはファンクション番号4EH(最初のファイルの検索)とファンクション番号4FH(次のファイルの検索)を用います．

具体的に説明しましょう．図2.1.2に処理の概要を表したフローチャートを示します．

はじめに,

```
dta = SPACE$(129)
```

によって,バッファDTAの内容をクリアしておきます.

```
fld = path$ + SPACE$(128 - LEN(pas$))
MID$(fld, LEN(path$) + 1) = CHR$(0)
```

この2行では,パス名をASCIIZ形式,つまりNULL(CHR$(10))で終わるASCII

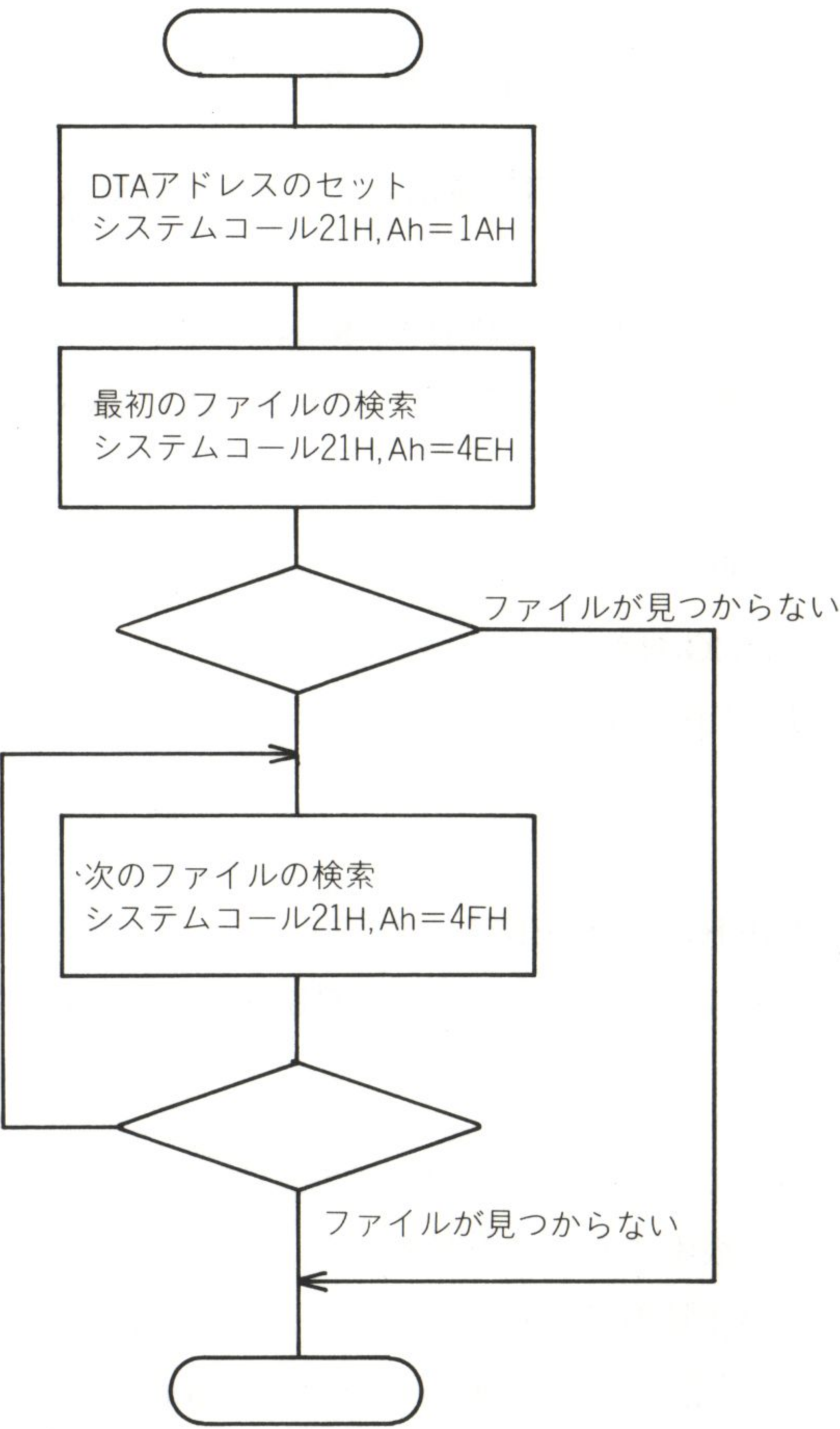
●図2.1.2 システムコールを利用したファイル検索のフローチャート

DTAアドレスのセット
システムコール21H, Ah=1AH

最初のファイルの検索
システムコール21H, Ah=4EH

ファイルが見つからない

次のファイルの検索
システムコール21H, Ah=4FH

ファイルが見つからない

文字列にしています.

ファンクション番号4EHを実行して，最初のファイルを検索する前に，ファンクション番号1AHのシステムコールにより,DTAアドレスの設定を行っておく必要があります.

具体的には,レジスタR.axに1AHを,R.dsとR.dxにDTAのセグメントとオフセットをセットしてから，CALL INTERRUPTX命令を実行します.

```
R.ax = &H1A00
R.ds = VARSEG(dta)
R.dx = VARPTR(dta)
R.es = -1
CALL INTERRUPTX(&H21, R, R)
```

次に，ファンクション番号4EH(最初のファイルの検索)のシステムコールを行います.

手順としては，レジスタR.axに4EHを，R.dsとR.dxにパス名のセグメントとオフセットを,R.cxに属性コードをセットしてからCALL INTERRUPTX命令を実行します.

```
R.ax = &H4E00
R.ds = VARSEG(fld)
R.dx = VARPTR(fld)
R.es = -1
R.cx = &H0
CALL INTERRUPTX(&H21, R, R)
IF (R.ax = &H12 OR R.ax = &H2 OR R.ax = &H3) THEN GOTO
    skip
```

属性コードは&H0がセットしてありますが，これは検索するファイルの属性が通常のファイルのみ検索するという意味です.

ファンクションが正常終了すると,DTAには表2.1.1に示すファイルに関する情報が収められます.正常に終了しなかった場合は,R.axにエラーコード(02H；ファイルがみつからない，03H；パスが見つからない，12H；もうファイルがない，一

●表2.1.1 DTAに得られる情報

オフセット	長さ（バイト）	値
00H	21	MS-DOSが次の段階でファンクション4FHをコールするときに使用する
15H	1	ファイルの属性
16H	2	最後の書き込みの時刻
18H	2	最後の書き込みの日付
1AH	2	ファイルサイズの下位ワード
1CH	2	ファイルサイズの上位ワード
1EH	13	ASCIIZ形式のファイル名と拡張子両者の間はピリオドが挿入される

致するものが見つからなかった)がセットされますので，以下の処理を中断して，行ラベルskipにジャンプさせます．

2つ目以降のファイルを続けて検索するために，ファンクション4FH(次のファイルを検索)のシステムコールを，ファイルが見つからなくなるか，ファイル数が200を越えるまで行います．プログラムではDO/LOOP UNTILループを使っています．

```
DO
    sumfile = sumfile + 1
    flag(sumfile) = MID$(dta, &H16, 1)
    file$(sumfile) = MID$(dta, &H1F, 12)
    tstmp(sumfile) = MID$(dta, &H17, 2)
    dstmp(sumfile) = MID$(dta, &H19, 2)

    MID$(dta, &H1F, 12) = SPACE$(12)
    MID$(dta, &H17, 2) = SPACE$(2)
    MID$(dta, &H19, 2) = SPACE$(2)

    R.ax = &H4F00
    R.dx = VARPTR(fld)
```

```
      R.ds = VARSEG(fld)
      R.es = -1
      R.cx = &H0
      CALL INTERRUPTX(&H21, R, R)
   LOOP UNTIL (R.ax = &H12) OR (sumfile >= 200)
```

まず,バッファDTAからそれぞれの配列にファイルの情報を代入します.次に,DTAから配列に代入した部分のみ,スペースに置き換えておきます.DTAの残りの情報はファンクション4FHを行う場合に必要になりますから,そのままにしておきます.

ファンクション番号4EHの場合とほぼ同じように,レジスタR.axに今度は4FHを,R.dsとR.dxにパス名のセグメントとオフセットを,R.cxに属性コードをセットしてからCALL INTERRUPTX命令を実行します.

この場合も,DTAには表2.1.1と同じ項目の情報が得られますので,最初のファイルの場合と同様の処理で,ファイル名や最後の書き込み日時などをDTAより切り出します.

ループの終了はR.axがエラーコード12Hを返すか,ファイル数が200を越えたところで起こります.

ところで,ファイル書き込みの日時は表2.1.2および表2.1.3のようにビット列で与えられるため,そのままでは処理できません.そこで**2.1.4.7**で示すビット列から文字列に変換するFUNCTIONプロシージャを作成して,これを利用することによって,処理しています.

なお,QBでシステムコールを実行する場合には,次のことにも注意を払わなければいけません.

(1) CALL INTERRUPTX命令はクイックライブラリQB.LIBに入っています.したがってこの命令を使うには,QB環境を起動する場合に,このライブラリを読み込んでおく必要があります.その方法として,次のようにMS-DOSのコマンドラインからQBコマンド投入時に,クイックライブラリの名前を指定せずに/Lオプションを使用すると自動的に読み込まれます.

```
QB /L
```

●表2.1.2 最後の書き込みの時刻のフォーマット

ビット	意味
0～4	秒／2
5～10	分（0～59）
11～15	時（0～23）

●表2.1.3 最後の書き込みの日付のフォーマット

ビット	意味
0～4	日
5～8	月
9～15	年－1980

（2） CALL INTERRUPTX命令を実際に使用する場合には，ヘッダファイルQB.BIをインクルードする必要があります．なおセットアッププログラムでQB環境を構築した場合，このQB.BIはQuick BASIC Ver.4.2では，サブディレクトリBINの中に入っていますが，Ver.4.5からは，サブディレクトリINCLUDEの中に入っているのでパス名を注意しなければいけません．

```
'$INCLUDE: 'a:¥INCLUDE¥QB.BI'
```

このように本プログラムでは，Ver.4.5用になっています．もし，Ver.4.2をお使いでしたら，パス名¥INCLUDEを¥BINに変更してください．またVer.4.5をお使いの人でも，サブディレクトリINCLUDEをAドライブ以外にセットアップしてある場合は，ドライブ名を書き直してください．

（3） CALL INTERRUPTX命令を使用する場合，メタコマンド$INCLUDEは，CALL INTERRUPTX命令が書かれているモジュールレベルコードに書かなければなりません．

2.1.4.2 SUBプロシージャ PR

このプロシージャの主な働きは，入力ファイルからテキストファイルがなくなるまで1行ずつ読み込んできて，行番号を付加して印字させるものです．1行の印字数が設定された数値より長い場合には，自動的に複数行に分けて印字するよ

うにします．またその際，“語”が分割されないように行を切断するようにしています．インデントがつけられた行を複数行に折り曲げて印字する場合は，そのインデントをくずさないようにしてあります．これによって，印字されたプログラムの可読性はかなり良くなるとおもわれます．

もうひとつここで注意したことは，QBのエディタは，ほかのBASICのエディタ(たとえばN_{88}-BASIC)と違って，1行が255文字あって，プログラムを入力していっても画面上で80文字で折れ曲がらず，横にスクロールしていきます．したがって，1行が長いプログラムをほかのエディタで入力していくと複数行になるものでも，QBのエディタ上ではあくまで1行となります．そのようなプログラムをこの清書プログラムで印字すると，当然複数行にわたって印字されますが，QBのエディタ上ではあくまで1行であるということを認識させておく必要があります．そこで，このプログラムでは，QB上で1行のプログラムを複数行に分けて印字した場合には，行を切断した部分の最後に，継続の印としてアンダースコア(_)をつけることにしました．また，その場合プログラムにつけたインデントをくずさない工夫もしました．

では，SUBプロシージャPRの具体的な説明にはいります．図2.1.3にこのプロシージャのフローチャートを示します．

（1） 入力と出力ファイルをそれぞれオープンします．

（2） 行番号を表す変数lnum，印字行数を表す変数lincnt，印字ページを表す変数pageの初期値をそれぞれ設定します．

（3） 1ページ目のタイトルを印字するSUBプロシージャTitlePrをコールします．タイトルはDOSのシステムコールによって得られたファイル名，ファイルの作成日時，印刷時の日時およびページです．

（4） 次にこのプロシージャの主要な作業ループに入ります．前判定のDO UNTILループの終了条件は，テキストファイルの終了を見つけたときになります．

（5） DOループ内では，最初にテキストファイルを1行分，入力ファイルから文字型配列prfile$に読み込みます．

SUBプロシージャProcTabを呼びます．

QBエディタは，エディト時にTABキーを押した場合，テキストファイルにはタブコード(&H9)を埋め込むのではなく，スペースを挿入します．しかし，QB以外のエディタでQBのソースプログラムを書いた場合，テキストファイル中にタブコ

●図2.1.3
SUB PRの
フローチャート

ファイルOPEN

lnum←0 lincnt←0

TitlPr …タイトル印刷

i←1

lnum←lnum+1

ファイルから
1行読み込む

ProcTab

行番号：を行の
先頭につける

LEN(prfile$(i))
>clnum

False

True

j←1

j≦i

False

True

LnPrint

lincnt←lincnt+1

lincnt
=rwnum-2

True

FFeed

j←j+1

False

行の折れ曲がり
位置キースペース

True

前方へスペースを捜す

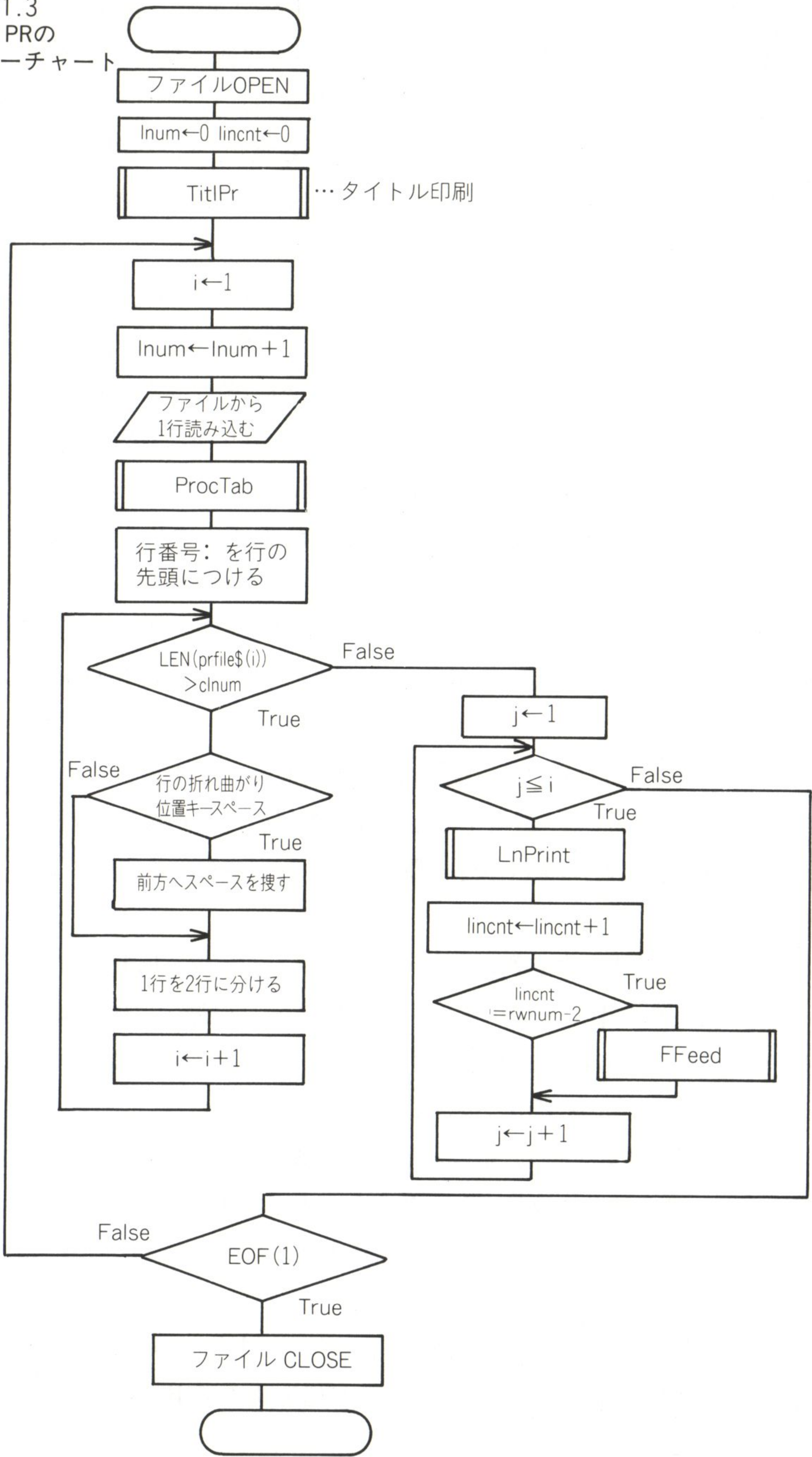
1行を2行に分ける

i←i+1

False

EOF(1)

True

ファイル CLOSE

ードが入っているかもしれません．プロシージャProcTabは，読み込んだテキストファイルにタブコードが存在した場合は，スペースに置き換える働きをします．

（6） 1行分のテキストファイルの先頭に行番号を付加します．その前にテキストファイルの最初の文字が，字下げされている場合，行を分割した場合にその情報が必要になるので，変数indentにあらかじめ求めておきます．そのときにはFUNCTIONプロシージャIndentNを使用します．こうすることによって，1行を複数行に分割して印字しても，インデントを崩す心配はなくなります．

（7） 取り込んだ1行が，印字幅の設定値clnumより長い場合には，前判定のDO WHILEループの中で複数行に折り曲げます．短い場合はこのループは実行されません．ループ内では，文字列のclnum番目で折り曲げるわけですが，ちょうどその位置が“語”の内部の場合，言い換えるとその位置がスペースでない場合は，そこで折り曲げると，リストが読みにくくなります．そこで，その位置から1文字ずつ前の位置がスペースかどうか調べていき，見つかったところが“語”の切れ目ですからそこで折り曲げます．プログラムでは無限DOループを使って，スペースが見つかったところで，EXIT DO文でループを抜けます．

折り曲げる位置が見つかったら，配列prfile$のi番目とi＋1番目に行を分けて代入します．i＋1番目のほうへ代入するときは，i番目の行の先頭の文字列のインデントを考慮します．また，i番目の行の最後には，行が継続する印であるアンダースコア(＿)を付加します．

次にi＋1行目の最初の文字列のインデントをプロシージャIndentNを使って調べておきます．iの値を1インクリメントしてDOループの最初に戻ります．折り曲げた次の行がさらにclnumより大きい場合は，さらに同じ処理を繰り返しますが，そうでない場合はこのループから抜けます．

（8） ループを抜けた時点でiの値は分割した行数を示しています．そこでFOR/NEXTループを使って分割した行を，1行ずつSUBプロシージャLnPrintを使って印字していきます．プロシージャLnPrintについては次の項で詳説します．

1ページに印刷している行数は変数lincntでカウントしていますので，設定初期値rwnumから2引いた値になったら，改ページ処理するSUBプロシージャFFeedを呼びます．

(9) テキストファイルの終わりを見つけるとこのDOループは終了しますから，ファイルをクローズしてこのSUBプロシージャを終わります.

2.1.4.3 SUBプロシージャ LnPrint

プロシージャLnPrintでは，プロシージャPRから引数oneline$で渡されてきた1行分の印字文字列の印字処理を行います.

この処理の概要は，1行の先頭から1文字ずつ順に調べていき，ある文字列が特定のキーワードの場合には強調文字で印字し，それ以外の場合はそのまま印字するものです.

しかし，例外があって，コメント行を示すアポストロフィのあとと，ダブルクォートにはさまれた文字列は，たとえキーワードであっても強調文字にしません.

図2.1.4にこのプロシージャのフローチャートを示します.処理の詳細は次のとおりです.

(1) 変数rf，qfおよびkflagは，現在調べている文字が，アポストロフィのあとか，ダブルクォートの間か，あるいは漢字コードにはさまれているかを表すフラグです．1行の処理に入る前に，すべてのフラグの内容をFalseに初期化しておきます.

(2) 一番外側のループは，前判定のDO WHILEループで処理します．1行の文字数をあらかじめ関数LENで求めておき，その文字数分印字処理が終わったら，このループを抜けて改行命令を出し，このプロシージャの処理を終わります.

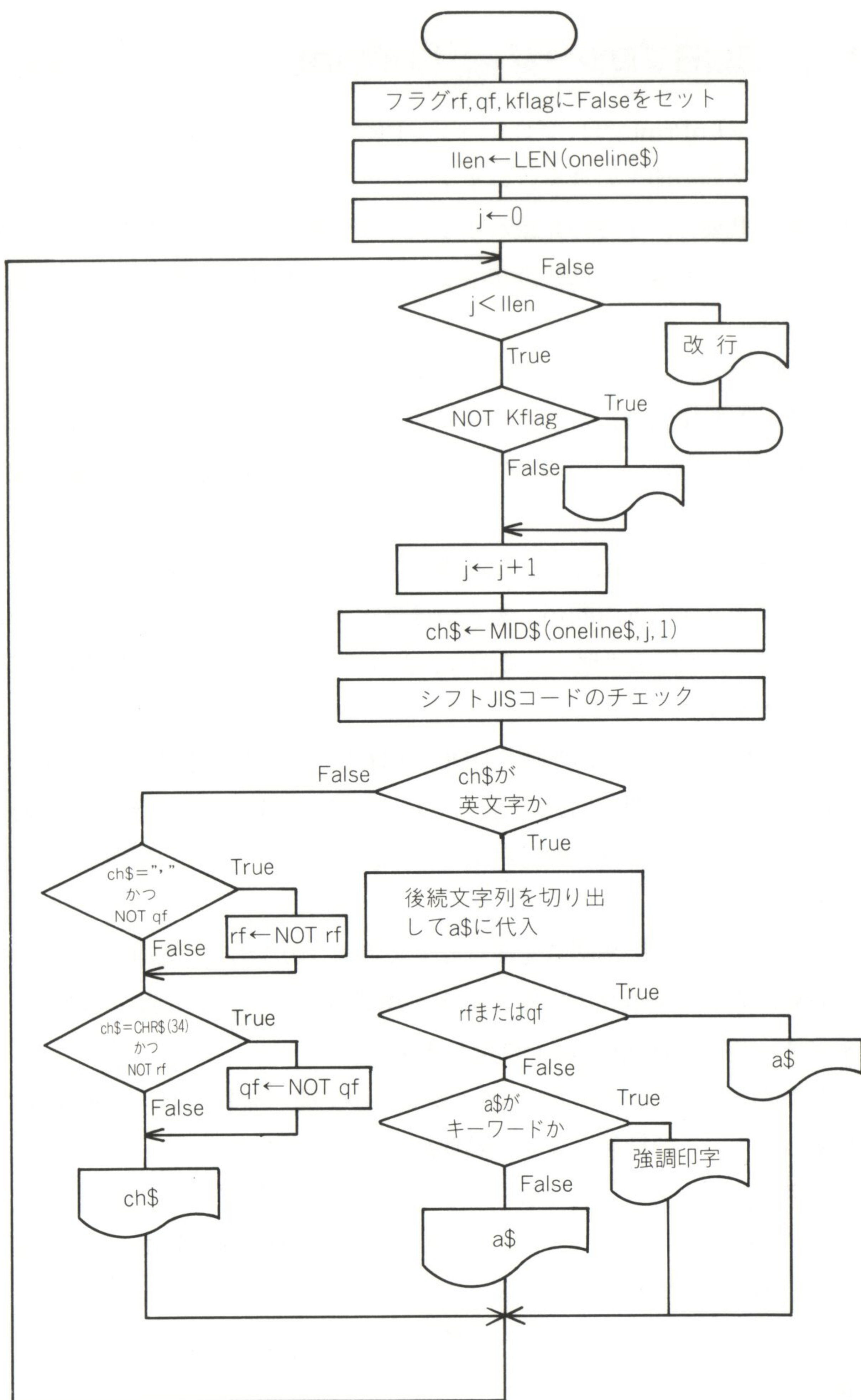
(3) ループ内では，MID$関数を使って1文字ずつ切り出して処理していきます.

最初に関数IsShiftJisで，その文字が漢字コードかどうか調べます．もし，漢字コードならkflagがTrueになります.

(4) 次にこの1文字がアルファベットかどうかを調べます.これには関数IsAlphaを使います．この結果によって処理が2つにわかれます.

(5) 文字がアルファベットでないときは，ただ単にその文字を印字します．その際，その文字がアポストロフィであり，かつフラグqfがFalseのときはフラグrfを反転しておきます.

同様に，文字がダブルクォートで，かつフラグrfがFalseのときは，フラグqfを反転します.

●図2.1.4 SUB LnPrintのフローチャート

調べた1文字がアルファベットの場合は，続く文字列の切り出しを行います．文字変数a$の内容をヌルにしておいて，後続の文字を次々と調べ，アルファベット，数字，型宣言文字以外の文字が現れるまで，a$に足しこんでいきます．

（6） フラグrfとqfを調べ，このどちらか，または両方がTrueの場合はa$をそのまま印字します．そうでない場合，すなわち両フラグともFalseのときは，a$がキーワードかどうか関数SearchKeyで調べます．キーワードの場合はプリンタの強調文字コードESC！と解除コードESC”にa$をはさんで印字します．

そうでないときは，ふつうの文字で印字します．このようなプログラムの中からのプリンタの制御については次項で説明します．

2.1.4.4 プログラムからのプリンタ制御

BASICのプログラムを書いていて，プリンタをソフトウェアでいろいろ制御したいと思うことがあります．たとえば，このプログラムでは，印字文字サイズをエリートサイズにするとか，特定の文字を強調文字で印刷するとかしていますが，これらはすべてBASICのプログラム中からプリンタを制御しています．

具体的には，各プリンタ固有のコントロールコードをマニュアルで調べ，PRINT文でそのコードをプリンタに送ってやります．

PC-PR201系のプリンタを使用してエリートサイズで印刷させるには，[ESC]＋[E]コードを送ってやります．このプログラムでは，

```
PRINT #2, CHR$(27) + "E"
```

のようにしています．ここでは印刷出力をファイル出力しているのでPRINT #2のような命令を使っていますが，印字にLPRINT文を使っている場合は，

```
LPRINT CHR$(27) + "E"
```

のように書きます．27はESCコードの値です．

また，ある文字列を強調文字で印刷する場合は，開始のコードが[ESC]＋[!]で，終了のコードが[ESC]＋[”]ですから，プログラムのように，もし文字列a$を強調文字で印刷したい場合は，その前後を開始と終了のコードではさんで，

```
PRINT #2, CHR$(27) + "!" ; a$; CHR$(27) + CHR$(34)
```

のように書きます．

したがって，PC-PR201系以外のプリンタを使って，このプログラムを利用する場合には，自分のプリンタのコードをマニュアルで調べてプログラム中のこれらの部分を書き直します．

2.1.4.5 FUNCTIONプロシージャ　SearchKey

この関数は，文字列を印刷するとき，その文字列がQBのキーワードかどうか検索するものです．

キーワードは，あらかじめ配列keyw$( )にアルファベット順に格納されており，調べたい文字列が検索できたときには，そのキーワードが格納されていた配列の添字を，見つからない場合は，その文字列に最も近いキーワードが格納されている配列の添字を返します．

文字列の検索法は2分法(バイナリサーチ)と呼ばれる手法を用いています．図2.1.5にプロシージャのフローチャートを示します．

2分法は，ソートされた配列要素を検索していく効率的な方法のひとつで，調べる範囲を順に，前後に2分していく方法です．

簡単な例を用いて2分法の概略を説明しましょう．

キーワードが8個だとして配列keyw$(0)～keyw$(7)に次のように格納されているとします．

keyw$	AS	CALL	DATA	END	IF	REM	STOP	TO
添字	0	1	2	3	4	5	6	7

これより変数hとtの初期値はそれぞれh＝0とt＝7となります．

まず，検索する文字列が“DATA”，つまりa$=“DATA”の場合について考えてみます．

このループは，後判定型のDO/LOOP UNTILで，ループの脱出条件は，h>tがTrueになったときです．

●図2.1.5 FUNCTION SearchKeyのフローチャート

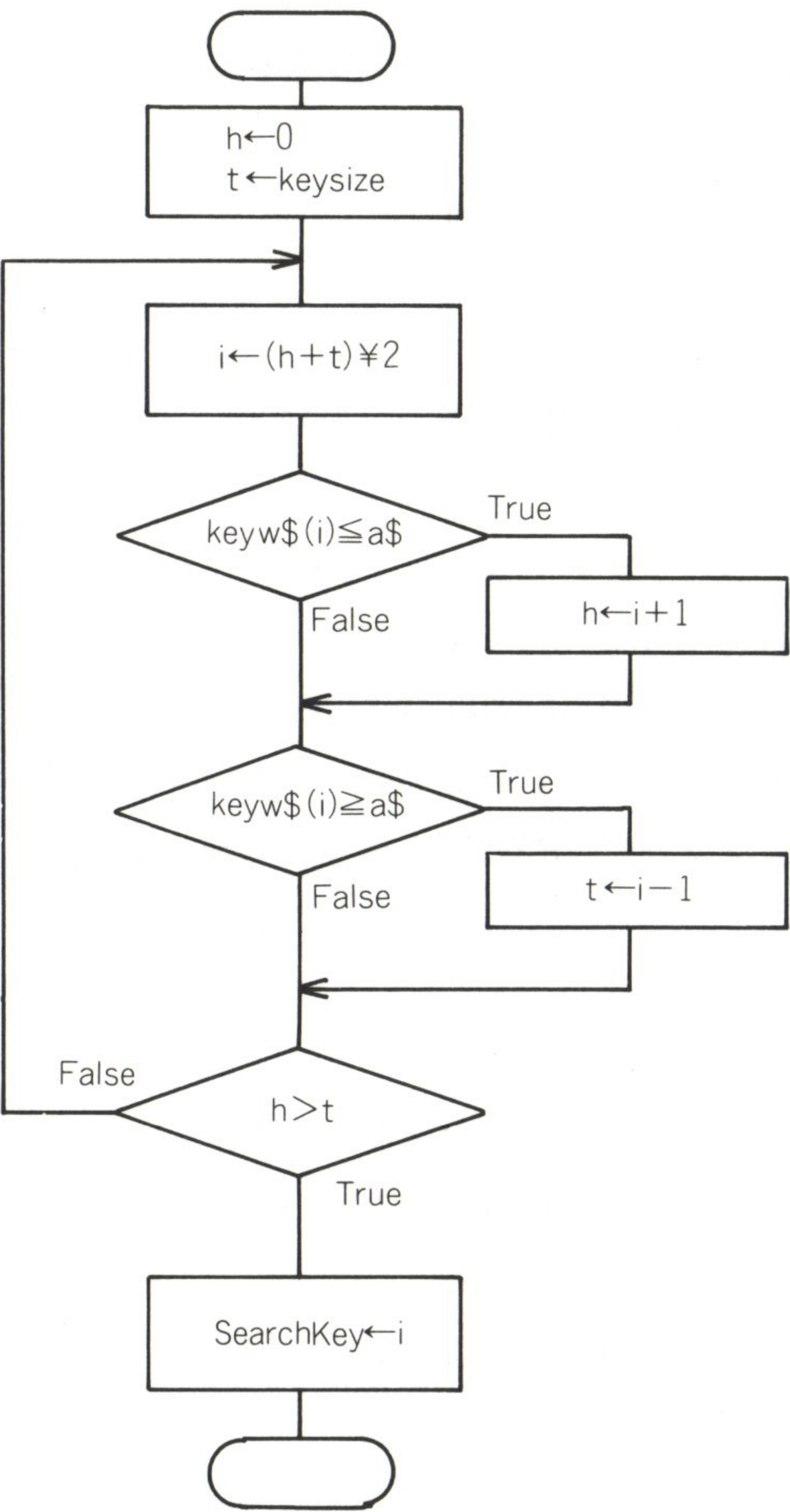
h←0
t←keysize

i←(h+t)¥2

keyw$(i)≦a$
True
h←i+1
False

keyw$(i)≧a$
True
t←i−1
False

h>t
False
True

SearchKey←i

1回目のループ

i ← （h ＋ t）¥ 2

ですから，i ← （0 ＋ 7）¥ 2 となって，iには3が代入されます．

したがって， keyw$(3)=“END” ≧ “DATA” ですから

t=2 となります．

h<t ですからループからは脱出しません．

2回目のループ

i ← （0 + 2）¥ 2　となって，iには1が代入されます．

したがって，　keyw$(1)=“CALL”≦“DATA” ですから

h＝2　となります．

h<t　ですからループからは脱出しません．

3回目のループ

i ← （2 + 2）¥ 2　となって，iには2が代入されます．

したがって，　keyw$(2)=“DATA”≦“DATA” ですから

h＝3　となります．

今度は，h>t　となりますからループを脱出します．

関数SearchKeyはiの値2がセットされて返されます．

このように検索する文字列が配列keyw$( )の中の要素と一致する場合は，その配列要素の添字が関数より返ってきますが，一致するものがない場合はどうでしょうか．そこで，次は文字列“NEXT”を検索する場合について考えてみます．

1回目のループ

i ← （h + t）¥ 2

ですから，i ← （0 + 7）¥ 2　となって，iには3が代入されます．

したがって，　keyw$(3)=“END”≦“NEXT” ですから

h＝4　となります．

h<t　ですからループからは脱出しません．

2回目のループ

i ← （4 + 7）¥ 2　となって，iには5が代入されます．

したがって，　keyw$(5)=“REM”≧“NEXT” ですから

t＝4　となります．

h<t　ですからループからは脱出しません．

3回目のループ

i ← （4 + 4）¥ 2　となって，iには4が代入されます．

したがって，　keyw$(4)=“IF”≦“NEXT” ですから

h＝5　となります．

今度は，h>t　となりますからループを脱出します．

関数SearchKeyはiの値4がセットされて返されます．

したがって，この場合は検索文字列とキーワードは一致しませんでしたが，検索した文字列に近いキーワードの配列要素の添字が返されたことになります．

2.1.4.6 SUBプロシージャ SelFile

このプロシージャは，配列file$( )中にあるファイル名の中から，カーソル操作により，1個のファイルを選択するものです．この中で，

```
a$ = CHR$(0, &H48)
```

のような，従来のBASICでは見られない文が現れます．これは，次のような理由によります．

PC-9801では，矢印キーは，2バイトのコードとして割りつけられています．上位バイトはゼロで，下位バイトは，48Hの場合[↑]キー，50Hの場合は[↓]キー，4BHの場合[←]キー，4DHの場合[→]キーです．この情報はマニュアル(Ver.4.5のHandbook 396ページ，ASCII文字コードおよびVer.4.2のStatement & Function Reference 432ページ，キーボードスキャンコード)に書かれています．

関数INKEY$は，2バイトコードのキー(漢字やファンクションキーなど)が押されると，2バイト文字として値を返すよう拡張されています．

関数CHR$( )も，2バイトの文字を表現できるように拡張されています．その引数はCHR$(上位，下位)となります．

2.1.4.7 そのほかのFUNCTIONプロシージャ

そのほかのおもな関数の働きについて簡単に説明しておきます．

FUNCTION IsDigit(c$)

パラメータc$が数字列かどうか判定します．

FUNCTION IsVar(b$)

パラメータb$が変数名と型宣言子からなりたっているかどうか判定します．

FUNCTION IsAlfa(a$)

パラメータa$が英文字だけで構成されているかどうか判定します．

FUNCTION IsShiftJis(ch$)

パラメータch$がシフトJISコードかどうか判定します.

FUNCTION　DispYear%(d%),
　　　　　DispMonth%(d%),
　　　　　DispDay%(d%)

MS-DOSのシステムコールで得られた，ファイルの作成年月日ビット情報パラメータd%から，整数値として年月日を得る関数です.

FUNCTION　DispHour%(d%),
　　　　　DispMint%(d%),
　　　　　DispSecd%(d%)

MS-DOSのシステムコールで得られた，ファイルの作成時分秒ビット情報パラメータd%から，整数値として時分秒を得る関数です.

2.1.5 実行ファイルの作成と使用方法

Quick BASICは，それ自身の環境内でプログラムの開発や実行ができますが，この清書プログラムのような性格のプログラムは，QBの環境から離れて，MS-DOSの外部コマンドのような使い方をすると便利です.

本節では，MS-DOSのコマンドラインから直接実行可能なEXEファイルの作成方法と，清書プログラムの使用方法を述べます.

まずEXEファイル作成方法を具体的に説明しましょう.

（1） QB環境内にプログラムFINELST. BASをロードします.

（2） メニューから[R/実行]を選びます．サブメニューが表示されますから[X/実行ファイルの作成]を選びます．ここで注意してほしいのは，メニューがFullメニューになっていることを確認してください．もしそうなっていないときは，メニューで[O/オプション]→[F/Fullメニュー]を選択しておきます.

（3） 実行ファイル作成画面が表示されます．ここで大切な確認事項は実行ファイル名と実行ファイルの型です．実行ファイル名は，ディフォルトでロードしてきたプログラムのファイル名に拡張子EXEをつけたものが表示されています．この例でいうと，FINELST. EXEと表示されているはずです．そのま

までよければ変更する必要はありませんが,変更したい場合は[TAB]キーでカーソルを移動させたのちファイル名を変更してください.

実行ファイルの型は2種類あって[X/ランタイム分離型]と[A/独立型]があります.ディフォルトはランタイム分離型ですが,DOSのコマンドのようにQBの環境から完全に分離させたいときは独立型を選択してください.すべてのオプションを決定したら[↵]キーを押します.コンパイルとリンクが行われEXEファイルが作成されます.

次に,作成したEXEファイルの使用方法について説明します.

(1) FINELST.EXEファイルが入ったフロッピィディスクをセットして,そのドライブをカレントドライブにします.

また,空いたドライブに,リストをとりたいテキストファイルが入ったディスクをセットします.

(2) 次にMS-DOSのコマンドラインからEXEファイルのファイル名を入力します.たとえば,カレントドライブがAドライブとすると,

A>FINELST

と入力します.

(3) 画面にリストしたいファイルのドライブとパス名を聞いてきますから答えます.Bドライブのルートディレクトリであれば,

B:

と入力します.Bドライブのサブディレクトリ LISTの中のファイルであれば,

B:¥LIST

と入力します.カレントドライブのカレントパスの場合は,何も入力しないで[↵]キーを押します.

(4) 画面にファイル名の一覧表示がでますから,リストしたいファイル名をカーソルキーを移動させて選び,[↵]キーを押すと印字が始まります.

2.2 N_{88}-BASICソースリストのQuick BASICへの変換

2.2.1 概　要

パーソナルコンピュータPC-9801シリーズ上で動くBASICとして広く使用されているのは，N_{88}-BASICです．これはマイクロソフト社のMBASIC 5に準じた仕様を持っていますが，分岐先にラベルが使用できる点が改良されています．

QBも同じマイクロソフト社のBASICであって，MBASICの上位互換性を持たせてあります．したがって，MBASICで書かれたプログラムは原則的にはQB上で動くと言えます．

しかし，せっかく，BASICからわずらわしい行番号がとれたのに，それらをそのまま残すのは，気分的に愉快ではありません．

また，N_{88}-BASICは，MBASICとも細部では仕様が異なるので，そのままでは動きません．

本章では，N_{88}-BASICで書かれてアスキーセーブされたソースプログラムを，QBで動くソースファイルに変換するトランスレータを作成してみます．とは言っても，N_{88}-BASICは独自に発展してきた部分もありますし，機種依存，たとえば画面制御とか，グラフィックスなどの部分は形式的には似ていても，その働きを解読しなくては正しく変換できませんので，適当なところでとどめておきます．その意味では，完全なトランスレータではなくて，変換支援ソフトです．

本章でのプログラミングの基礎は，主として，N_{88}-BASICの構文を解析し，それをQBのものに置き換える点にあります．そのための考え方や，実際に作成されたモジュールについて説明します．

2.2.2 仕様

はじめに，仕様について検討します．そのためにはN88-BASICとQBの比較をしてみます．

	N88-BASIC	QB
行番号	必　要	なくても可
ラベル	可　　注1)	可　　注2)
予約語	引数の与え方が両者で異なるもの SCREEN, BEEP, LOCATE, WIDTH, CLS	
	CONSOLE	VIEW　PRINT
名　前	．(小数点)は名前の一部として使用可	．はフィールドを表す．

注1) N88-BASIC のラベルの使い方

```
100  GOTO *FLAG1
110  ........
120  ........
200  *FLAG1
210  ........
```

注2) QBのラベルの使い方

```
GOTO FLAG2
........
........
FLAG2:
........
```

このように，QBとN88-BASICの最も大きな違いは，行番号の有無と行ラベルの書き方です．

N88-BASICは，QBに比較して，構造的な制御文がきわめて少ないため，どうしてもGOTO，GOSUB文が多用されることになります．したがって，QBでは行番号が不用ですが，N88-BASICではGOTO，GOSUBの行き先が行番号で指定されていますので，その行番号を不用意に取り去るようなことがあってはなりません．

また，行き先が行ラベルで指定されていても，N88-BASICの行ラベル書式をそのまま使えませんから，QBではすべての行ラベルを書き換える必要があります．

これらの作業は，機械的にできることではありますが，N_{88}-BASICのようにGOTOやGOSUBが多いプログラムでは，ひとつのもれもなく，人の手によってファイルを書き換えるのは，かなり骨のおれる仕事といえるでしょう．

N_{88}-BASICとQBでは，画面制御やグラフィックス関係も細かく仕様が異なります．しかし，こちらは機械的に変換というわけにはいきません．たとえば，LOCATE文ひとつとっても，N_{88}-BASICでは，

LOCATE　X座標，Y座標　　　(X，Y＞＝0)

のものを，QBでは，

LOCATE　Y座標，X座標　　　(Y，X＞＝1)

にしなければなりません．

これだけ見ていると，X座標とY座標を交換して１加えるだけで，変換は簡単にすみそうですが，実際にはそうはいきません．

XとY座標が数値で与えられている場合はよいのですが，数式で与えられていると，いちどに処理が複雑になってしまいます．

そこで今回のコンバータは，これら画面制御やグラフィックス関係の変換は一切行わず，処理は機械的に行えるがかなり面倒なうえ，変換もれを起こしやすい不用な行番号削除，および行ラベルのつけ変え処理を行うものとします．

なお，変数名中のピリオドの扱いですが，先ほどの表からもあきらかなように，N_{88}-BASICとQBでは意味が異なります．N_{88}-BASICではユーザ定義型がないので，フィールドという概念がないことと，QBにおいては単純変数名にピリオドが含まれていてもエラーを生じないことから，そのままとしました．

もうひとつ，N_{88}-BASICがQBと大きく異なるものにエディタがあります．N_{88}-BASICのエディタは，１行が長くなるとディスプレイ上で見にくくなるので，それを防ぐために行の途中にLFコードを挿入して，改行ができるようになっています．しかし，QBのエディタでは，このLFコードを行の終わりと判断してしまうために，１行を複数行と判別してしまいます．したがって，コンバートを行う場合，１行のテキストの中にLFを見つけたら，そのコードを取り去り，そのあとのスペースを除去する処理が必要になります．

以上，仕様をまとめると，

（1） THEN, ELSE, GOTO, GOSUB文で行き先に指定された行番号を除いて，残りの行番号を取り去る．

（2） 行ラベルがあったら，QBの書式に書き換える．

（3） 1行の途中にLFコードがある場合，それを取り去り不用のスペースを除く．

（4） コメント行や，ダブルクォートにはさまれた文字列中は，変換作業をしない．

となります．

2.2.3 プログラム概要

図2.2.1にプログラムのゼネラルフローを示します．

コンバートプログラムを作成する場合，そのひとつの方法として，文字列配列変数を用意しておいて，変換前のソースファイルをいったんすべてそこへ読み込んで，変換処理を行うことが考えられます．

しかし，この方法だと変換するソースファイルと同じ大きさの配列を用意しなければならないため，変換できるファイルの大きさはパソコンのメモリの大きさに依存してしまいます．

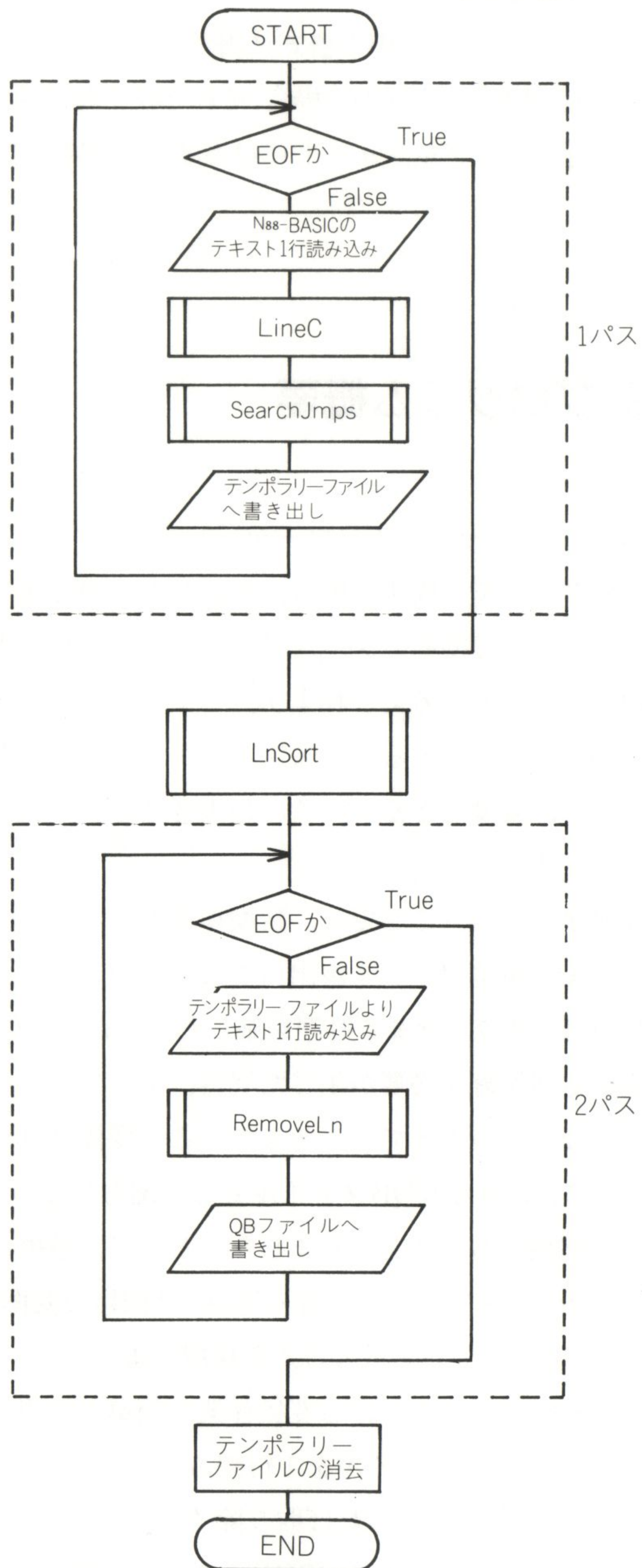
その問題に対処するため，ここでは，変換作業をテキストの1行単位で行い，作業が終了したら1行単位でファイル出力する方法をとりました．

この方法の欠点は，最後の行まで処理を施したあとに，その結果によって再び，最初の行から別の処理を施す必要が生じた場合，またファイルの読み込みと書き出しをしなければならないことです．そこで，この変換プログラムでは，1回目の作業が終了したら，いきなりQBファイルとして出力しないで，いったんテンポラリーファイルに格納する方法をとりました．そして，最後の行まで作業が終了したら，次はテンポラリーファイルを読み込み，2回目の変換処理を行ったあと，QBファイルとして出力する，2パス方式を採用しました．

それでは，それぞれのパスにおける変換作業を簡単に説明します．1パス目では，N_{88}-BASICのテキストファイルを1行読み込んで，

（1） 行の途中にLFコードがあれば取り除く．

（2） THEN, ELSE, GOTO, GOSUBのあとに行番号が続いた場合は，その

●図2.2.1 ゼネラルフローチャート

行番号を配列に格納しておく．

（3） THEN,ELSE,GOTO, GOSUBのあとに行ラベルが続いた場合は，QBの行ラベル書式に変換する．

といった処理を行ったあと，1行ずつフロッピィディスク上のテンポラリーファイルに書き出していきます．これをテキストファイルがなくなるまで行います．

次に2パス目に移る前に,行番号を格納した配列の内容を番号の小さい順にソートし，さらに行番号が重複しないようにします．

2パス目では,1パス目でいったん格納したフロッピィディスクのテンポラリーファイルから，逆に1行ずつテキストファイルを読み出して，

（1） 読み出した行の行番号と配列に格納してある行番号を順に比較していき，一致する行番号が存在した場合は，その行番号は消去しない．一致するものがない場合は，その行番号を消去してスペースに置き換える．

（2） その行がN_{88}-BASICの行ラベル(＊ラベル名)であった場合は,QBの行ラベル(ラベル名：)に置き換える．

といった処理を行ったあと，1行ずつ，指定した出力ファイルに書き出していきます．これをテンポラリーファイルの中のテキストファイルがなくなるまで行います．最後に，テンポラリーファイルを消去して，処理は終了します．

2.2.4 主要なプロシージャの説明

リスト2.2.1にプログラムの全リストを示します．

プログラムの中の主要なプロシージャについて順に説明していきます．

2.2.4.1 モジュールレベルコード

モジュールレベルコードでは，最初に，N_{88}-BASICのプログラムが格納された入力ファイル名と,QBに変換したプログラムの出力ファイル名の入力要求を行い，それぞれinpdev$とoutdev$に代入します．

モジュールレベルコード中心部では,最初のDO UNTIL/LOOPが1パス目の処理を，次のDO UNTIL/LOOPが2パス目の処理を行っています．両パスの間で，配列のソートなどを行うSUBプロシージャLnSortをコールします．

●リスト 2.2.1 プログラムFILECONV. BASの全リスト

```
 1: DECLARE SUB TakeWord (Cond!, OneLine$, ch$, a$, j!, llen!)
 2: DECLARE SUB LineC (OneLine$)
 3: DECLARE SUB RemoveLn (OneLine$, lnum!(), lp!)
 4: DECLARE SUB LnSort (lnum!(), lnp!)
 5: DECLARE FUNCTION IsDigit! (c$)
 6: DECLARE FUNCTION IsVar! (b$)
 7: DECLARE FUNCTION IsAlpha! (a$)
 8: DECLARE SUB SearchJmps (OneLine$, lnum(), lp)
 9: DECLARE SUB LnMem (OneLine$, n, l, lnum(), lp, af)
10:
11:
12: DIM lnum(100)
13:
14: CONST False = 0, True = NOT False
15:
16:  CLS
17:
18:  INPUT "Ｎ８８ＢＡＳＩＣのファイル名を入力してください．"; inpdev$
19:  INPUT "変換後のＱＢファイル名を入力してください．"; outdev$
20:
21:  dummy$ = "A:dummy"
22:
23:
24:  OPEN inpdev$ FOR INPUT AS #1
25:  OPEN dummy$ FOR OUTPUT AS #2
26:
27:  lp = 1
28:
29:  DO UNTIL EOF(1)
30:
31:     LINE INPUT #1, OneLine$
32:
33:     CALL LineC(OneLine$)
34:
35:     CALL SearchJmps(OneLine$, lnum(), lp)
36:
37:     PRINT #2, OneLine$
38:
39:  LOOP
40:
41:  lp = lp - 1
42:
43:  CLOSE #1, #2
44:
45:  CALL LnSort(lnum(), lp)
46:
47:
48:  OPEN dummy$ FOR INPUT AS #1
49:  OPEN outdev$ FOR OUTPUT AS #2
50:
51:  DO UNTIL EOF(1)
52:
53:     LINE INPUT #1, OneLine$
54:
55:     CALL RemoveLn(OneLine$, lnum(), lp)
56:
57:     PRINT OneLine$
58:     PRINT #2, OneLine$
59:
60:  LOOP
61:
62:  CLOSE #1, #2
63:
64:  KILL "A:dummy"
65:
66: END
```

```
 67:
 68: FUNCTION IsAlpha (a$)
 69:
 70:     IF ((a$ >= "A" AND a$ <= "Z") OR (a$ >= "a" AND a$ <= "z")) THEN
 71:         IsAlpha = True
 72:     ELSE
 73:         IsAlpha = False
 74:     END IF
 75:
 76: END FUNCTION
 77:
 78: FUNCTION IsDigit (c$)
 79:
 80:     IF (c$ >= "0" AND c$ <= "9") THEN
 81:         IsDigit = True
 82:     ELSE
 83:         IsDigit = False
 84:     END IF
 85:
 86: END FUNCTION
 87:
 88: FUNCTION IsVar (b$)
 89:
 90:     IF ((b$ >= "A" AND b$ <= "Z") OR (b$ >= "a" AND b$ <= "z") OR (b$ >= "0" AND b$ <=
        "9") OR b$ = "." OR b$ = "!" OR b$ = "#" OR b$ = "$" OR b$ = "%") THEN
 91:         IsVar = True
 92:     ELSE
 93:         IsVar = False
 94:     END IF
 95:
 96: END FUNCTION
 97:
 98: SUB LineC (OneLine$)
 99:
100:     lc = INSTR(OneLine$, CHR$(10))
101:
102:     DO WHILE lc <> 0
103:        sc = 1
104:        ch$ = MID$(OneLine$, lc + sc, 1)
105:        DO WHILE ch$ = " "
106:            sc = sc + 1
107:            ch$ = MID$(OneLine$, lc + sc, 1)
108:        LOOP
109:        OneLine$ = LEFT$(OneLine$, lc - 1) + " " + MID$(OneLine$, lc + sc)
110:
111:        lc = INSTR(OneLine$, CHR$(10))
112:
113:     LOOP
114:
115: END SUB
116:
117: SUB LnMem (OneLine$, j, llen, lnum(), lp, af)
118:
119: DO WHILE j < llen
120:     j = j + 1
121:     ch$ = MID$(OneLine$, j, 1)
122:
123:     IF ch$ = ":" THEN EXIT DO
124:
125:     IF IsDigit(ch$) THEN
126:
127:         Cond = 1
128:         CALL TakeWord(Cond, OneLine$, ch$, a$, j, llen)
129:
130:         lnum(lp) = VAL(a$)
131:
132:         lp = lp + 1
133:
134:         IF af THEN EXIT DO
```

```
135:
136:     ELSEIF (ch$ = "*") THEN
137:
138:         Cond = 4
139:         CALL TakeWord(Cond, OneLine$, ch$, a$, j, llen)
140:
141:         wlen = LEN(a$)
142:         a$ = RIGHT$(a$, wlen - 1) + " "
143:         OneLine$ = LEFT$(OneLine$, j - wlen) + a$ + MID$(OneLine$, j + 1)
144:
145:         IF af THEN EXIT DO
146:
147:     ELSE
148:         IF IsAlpha(ch$) THEN
149:
150:             Cond = 2
151:             CALL TakeWord(Cond, OneLine$, ch$, a$, j, llen)
152:
153:             IF a$ = "GOTO" OR a$ = "GOSUB" THEN
154:                 af = True
155:                 CALL LnMem(OneLine$, j, llen, lnum(), lp, af)
156:                 EXIT DO
157:             ELSE
158:                 EXIT DO
159:             END IF
160:         END IF
161:     END IF
162: LOOP
163:
164: END SUB
165:
166: SUB LnSort (lnum(), lnp)
167:
168: ' IF lnum(1) = 0 THEN EXIT SUB
169:
170:  DO
171:     change = False
172:     FOR k = 1 TO lnp - 1
173:         IF lnum(k) > lnum(k + 1) THEN
174:             SWAP lnum(k), lnum(k + 1)
175:             change = True
176:         END IF
177:     NEXT k
178:  LOOP UNTIL NOT change
179:
180:  i = 1
181:  DO
182:     IF lnum(i) = lnum(i + 1) THEN
183:         FOR j = i TO lnp - 1
184:             lnum(j) = lnum(j + 1)
185:         NEXT j
186:         lnp = lnp - 1
187:     ELSE
188:         i = i + 1
189:     END IF
190:  LOOP UNTIL i = lnp
191:
192: END SUB
193:
194: SUB RemoveLn (OneLine$, lnum(), lp)
195:
196:  llen = LEN(OneLine$)
197:  j = 0
198:
199:  '------- プロセス1 ------
200:
201:  DO WHILE j < llen
202:     j = j + 1
203:     ch$ = MID$(OneLine$, j, 1)
```

```
204:        IF IsDigit(ch$) THEN
205:
206:            Cond = 1
207:            CALL TakeWord(Cond, OneLine$, ch$, a$, j, llen)
208:
209:            EXIT DO
210:        END IF
211:    LOOP
212:
213:    an = VAL(a$)
214:    numflag = False
215:    FOR i = 1 TO lp
216:        IF an = lnum(i) THEN
217:            numflag = True
218:            EXIT FOR
219:        END IF
220:    NEXT i
221:
222:    IF NOT numflag THEN
223:        OneLine$ = STRING$(LEN(a$), " ") + MID$(OneLine$, j + 1)
224:    END IF
225:
226:    '------- プ ロ セ ス 2  ------
227:
228:    DO WHILE j < llen
229:        j = j + 1
230:        ch$ = MID$(OneLine$, j, 1)
231:        IF ch$ <> " " THEN
232:
233:            IF ch$ = "*" THEN
234:
235:                Cond = 3
236:                CALL TakeWord(Cond, OneLine$, ch$, a$, j, llen)
237:
238:                wlen = LEN(a$)
239:
240:                a$ = RIGHT$(a$, wlen - 1) + ":"
241:                OneLine$ = LEFT$(OneLine$, j - wlen) + a$ + MID$(OneLine$, j + 1)
242:
243:                EXIT DO
244:            ELSE
245:                EXIT DO
246:            END IF
247:        END IF
248:    LOOP
249:
250: END SUB
251:
252: SUB SearchJmps (OneLine$, lnum(), lp)
253:
254: rf = False
255: qf = False
256:
257: llen = LEN(OneLine$)
258:
259: j = 0
260: DO WHILE j < llen
261:     j = j + 1
262:     ch$ = MID$(OneLine$, j, 1)
263:     IF IsAlpha(ch$) THEN
264:
265:         Cond = 3
266:         CALL TakeWord(Cond, OneLine$, ch$, a$, j, llen)
267:
268:         IF NOT (rf OR qf) THEN
269:
270:             IF a$ = "REM" THEN
271:                 rf = NOT rf
```

```
272:                GOTO L1
273:            END IF
274:
275:            IF a$ = "GOTO" OR a$ = "GOSUB" OR a$ = "THEN" OR a$ = "ELSE" THEN
276:                af = False
277:                IF a$ = "THEN" OR a$ = "ELSE" THEN af = True
278:                CALL LnMem(OneLine$, j, llen, lnum(), lp, af)
279:            END IF
280:        END IF
281:    ELSE
282:
283:        IF ((ch$ = "'") AND (NOT qf)) THEN rf = NOT rf
284: L1:
285:        IF ((ch$ = CHR$(34)) AND (NOT rf)) THEN qf = NOT qf
286:    END IF
287: LOOP
288:
289: END SUB
290:
291: SUB TakeWord (Cond, OneLine$, ch$, a$, j, llen)
292:
293:        a$ = ""
294:        DO
295:            a$ = a$ + ch$
296:            j = j + 1
297:            ch$ = MID$(OneLine$, j, 1)
298:            SELECT CASE Cond
299:                CASE 1
300:                    IF (NOT IsDigit(ch$) OR (j > llen)) THEN EXIT DO
301:                CASE 2
302:                    IF (NOT IsAlpha(ch$) OR (j > llen)) THEN EXIT DO
303:
304:                CASE 3
305:                    IF (NOT IsVar(ch$) OR (j > llen)) THEN EXIT DO
306:                CASE 4
307:                    IF (NOT (IsDigit(ch$) OR IsAlpha(ch$) OR (ch$ = ".")) OR (j > llen))
                    THEN EXIT DO
308:            END SELECT
309:        LOOP
310:        j = j - 1
311:
312: END SUB
313:
```

1パス目では，LINE INPUT文でN_{88}-BASICのテキストファイル1行を，文字列変数OneLine$に読み込みます．

SUBプロシージャLineCとSearchJmpsをコールして，1パス目の処理を行ったあと，テンポラリーファイルであるA：DUMMYに1行書き込みます．

これをN_{88}-BASICのテキストファイルがなくなるまで行います．

1パス目が終了したら，ファイルをクローズします．

2パス目に移る前に，配列の処理を行うSUBプロシージャLnSortをコールします．

2パス目ではテンポラリーファイルから1行読み込み，SUBプロシージャRemoveLnをコールします．

2.2.4.2 SUBプロシージャ LineC

1行の中に，もしLFコード(CHR$(10))を見つけたら，そのコード自身とそのコードの後方のスペースをすべて除去する処理をします．

これはN_{88}-BASICのエディタでは，プログラムを画面上で見やすくするために，LFコードを入れて見かけ上改行していますが，QBでは，そのままにしておくと，LFが挿入された行は1行と見なされないという問題が起こるからです

このプロシージャで処理すれば，1行にいくつLFコードが存在してもすべて取り除くため，1行が複数行に化けるといったことを避けることができます．

2.2.4.3 SUBプロシージャ SearchJmps

まず1行のテキストラインよりGOTO，GOSUB，THEN，ELSEの4つのキーワードの切り出しを行います．

rfとqfの2つのフラグは，**2.1**の清書プログラムで用いたものと同じで，rfはコメント行の，qfはダブルクォートにはさまれた文字列に対するチェックを行っています．

コメント行は，キーワードREMの場合と，アポストロフィ(')の場合の両方検知できます．

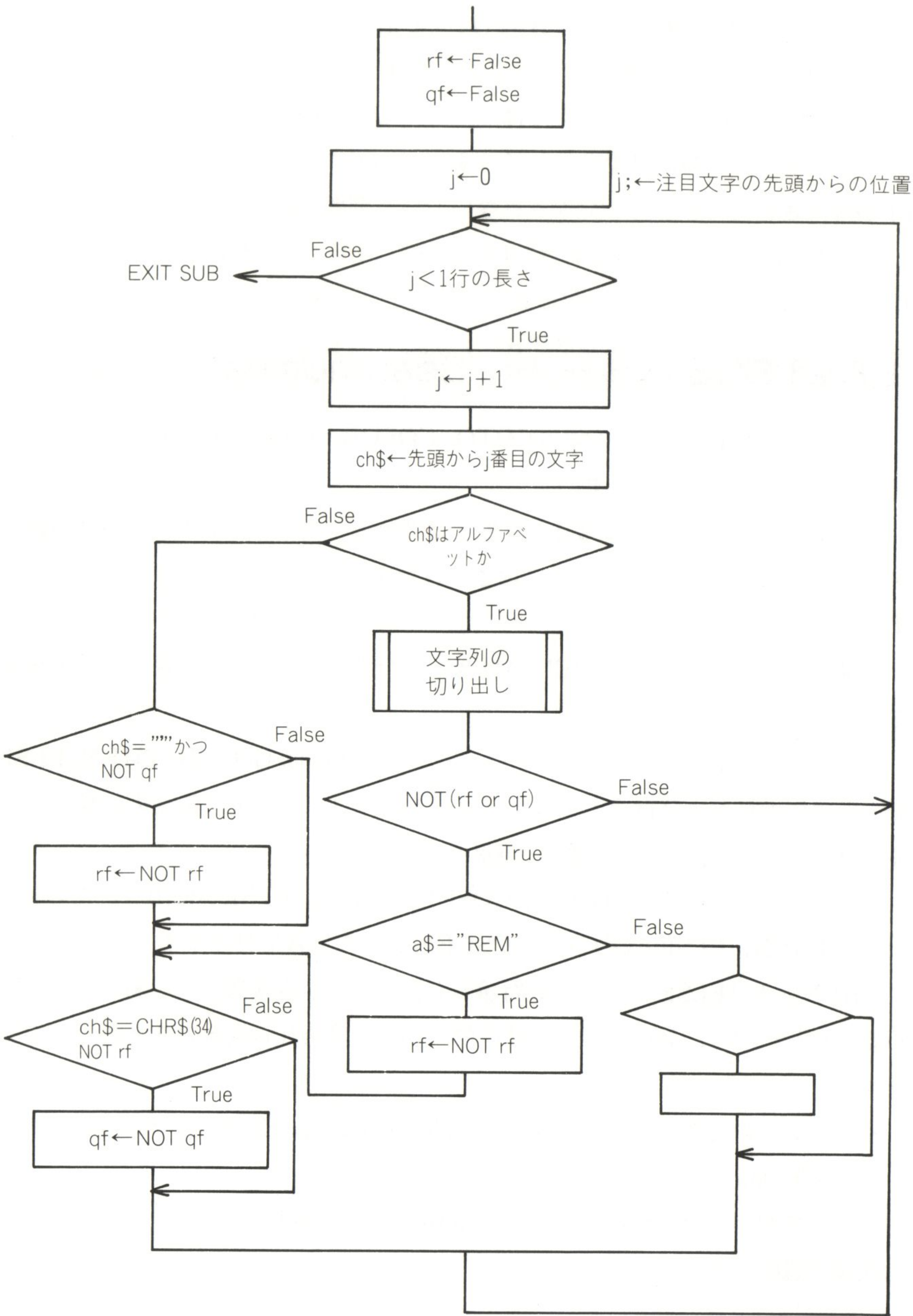
コメント行の中，およびダブルクォートの中のGOTO，GOSUB，THEN，ELSEのあとの行番号および行ラベルは，プログラムのコンバートの場合の対象行ラベルより除外します．rfとqfの働きを図2.2.2に示します．

もし，4つのキーワードGOTO，GOSUB，THEN，ELSEが検出されたら，それらをTHEN，ELSEとGOTO，GOSUBの2つの場合に分けます．

THENおよびELSEのあとには，行番号(行ラベル)が直接置かれる場合とGOTO 行番号(行ラベル)，GOSUB 行番号(行ラベル)が置かれる場合の2とおりがあります．

そこであとの処理で区別が必要なため，THENかELSEが検出された場合は，フラグafをTrueにしておきます．

そうしておいて，SUBプロシージャLnMemをコールします．このプロシージャは次項で説明します．

●図2.2.2 SUB Search Jmpsのフローチャート(rfとqfを中心に)

2.2.4.4 SUBプロシージャ LnMem

このプロシージャは,

（1） GOTOやGOSUB文などで,ジャンプ先に指定された行番号または行ラベルを配列に格納する.

（2） もし,行ラベルの場合はQBの書式にしたがって,ラベルの先頭のアスタリスクを取り去り,ラベル名のうしろにスペースをおく.

といった処理を行います.

具体的には,まずテキストラインから1語切り出します.

もしコロン(:)であれば,マルチステートメントにおける文の区切りですから,DOループより抜けたあと,SUBプロシージャからも抜けます.

もし先頭が数字ならば行番号と考えられるので,そのあとの連続した数字を切り出します.行番号は文字列として切り出されるので,関数VALを使って数値に変換したあとに,配列Lnumのlp番目にしまい,lpの値を1増加しておきます.もしフラグafがTrueならこのループを抜けます.

4つのキーワードTHEN, ELSE, GOTO, GOSUBのあとの先頭がアスタリスク(*)のときは,行ラベルと考えられますから,つづいてラベル名の切り出しを行います.ラベルの切り出しが終わったら先頭のアスタリスク(*)を取り去り,ラベルのうしろにスペースを挿入します.もしフラグafがTrueならこのループを抜けます.

最後に,英字だけからなる文字の検出をします.

THEN, ELSEの直後は行番号(ラベル),GOTO文,GOSUB文,さらにそのほかの実行文がくる可能性があります.このELSE節では,その中でGOTO文かGOSUB文がきたときにSUBプロシージャLnMem,すなわち自分自身のプロシージャを再帰的に呼び出します.その場合,フラグafはTrueにセットしておきます.

ここでフラグafについて説明しておきましょう.

IF文には,

```
IF 条件 THEN 行番号
```

という場合と,

```
IF 条件 THEN GOTO 行番号
```

という場合があります．

またGOTOとGOSUB文には，

```
ON GOTO 行番号1(行ラベル1), 行番号2(行ラベル2), …,
行番号n(行ラベルn)
```

といった複数の分岐先を示す場合があります．

したがって，THENとELSEの場合，その直後に行番号がくる場合は１個に限られますので，それ以後を調べる必要がありません．またGOTOとGOSUB文について考えても，THENやELSEのあとに続いた場合も，とりうる行番号はひとつだけですのでそれ以後を調べなくてもよいわけです．したがって，この条件をわける意味でフラグafを用意しておき，いま述べた場合にafをTrueにしておくことにより，行番号(行ラベル)をひとつだけ調べ，それを配列に格納し終わったら，SUBプロシージャLnMemを抜けるようにしておきます．

2.2.4.5 SUBプロシージャ　RemoveLn

このサブルーチンは，１行のテキストラインを受け取り，１パス目で，GOTOやGOSUB文などで指定された行番号を残して，行番号を取り去る処理をします．さらに，行ラベルがN_{88}-BASICとQBでは書式が異なるので，その処理も行います．

まず，SUBプロシージャTakeWordをコールして，１行のテキストラインより行番号を文字列として切り出し，組み込み関数VALを使って数値に変換します．そしてnumflagをFalseにしておいて，１パス目の処理で，消去不能な行番号を格納した配列lnum( )の内容と比較していきます．

もし，一致する行番号が存在した場合には，フラグを反転してループを抜けます．そして，フラグnumflagがFalse，つまりGOTO文などの行き先になっていなくても，プログラムの制御に関係ない行番号は削除して，その代わりにスペースに置き換えます．

次に行ラベルの変更を行います．つまり，N_{88}-BASICの行ラベルは“*ラベル名”ですが，QBの行ラベルは“ラベル名：”ですので，変換します．

この変換もSUBプロシージャTakeWordを使って，アスタリスク(*)ではじま

るラベル名を検索し，もし存在すればアスタリスクを取り去り，ラベル名の最後にコロン(：)をつけ加えます．

2.2.4.6 SUBプロシージャ TakeWord

1行のテキストラインから特定の文字列を切り出すプロシージャです．与えるパラメータとしては，1行のテキストラインがおさめられた文字列oneline$，テキストラインの注目文字列ch$とその先頭からの文字数j，テキストラインの文字数llen，切り出す文字列の条件区分を示すCondがあります．

これにより，パラメータa$に条件Condによって検出された文字列が代入されたあと，返されます．また，jも書き換わります．

条件Condは，このプロシージャの場合1から4までの数字をとり，1が数字だけからなる文字列，2が英文字だけからなる文字列，3が変数名と考えられる文字列，4がラベル名と考えられる文字列に対応しています．

2.2.4.7 SUBプロシージャ LnSort

このプロシージャは，1パス目の処理で，GOTO文などの行き先に指定された行番号が格納されている配列要素を，2パス目で処理しやすいように行番号の小さい順に並べ換え，さらに重複する行番号は取り除きます．

2.2.5 プログラムの実行方法

プログラムを実行させると，N88-BASICのファイル名を聞いてきますから，入力します．

次に，変換後のQBファイルの出力先を聞いてきますから，ファイル名を答えると，変換の作業が始まります．

実行例として，リスト2.2.2に示した，N88-BASICで書かれた待ち行列のシミュレーションプログラムをQBに変換したものを，リスト2.2.3に示します．

このプログラムは，グラフィックスや画面制御の命令を使っていませんから，変換されたQBプログラムは，ただちに実行可能です．

なお，この変換プログラムを使う場合，とくに注意しなければいけない点は，

（1） テンポラリーファイルをAドライブに作成するので，Aドライブに，変換前のN_{88}-BASICのソースファイルとほぼ同じ大きさのディスクの空きが必要なこと．

（2） 変換するN_{88}-BASICのソースファイルは，アスキーセーブされていなければいけないこと．

の2点です．

●リスト 2.2.2 N_{88}-BASICのソースファイル例

```
 1: 10 '  save "henkaten.n88",a          '25-Sep-89
 2: 20 '
 3: 1000 '----  待ち行列のシミュレーション（変化点方式） ----
 4: 1010    DIM TQ(500)
 5: 1020 '
 6: 1030    TEND = 50000!              'シミュレーション時間
 7: 1040    A = 12                     '平均到着時間間隔
 8: 1050    H = 10                     '平均サービス時間
 9: 1060 '
10: 1070    RANDOMIZE
11: 1080    TW = 0: STIME = 0: TT = 0: TQ(0) = TT
12: 1090    Q = 1: TQ(1) = TT - A * LOG(RND)
13: 1100    T1 = TT - H * LOG(RND): T2 = TQ(1)
14: 1110 '
15: 1120    WHILE TT <= TEND
16: 1130        IF T1 > T2 THEN 1180
17: 1140            IF Q <= 1 THEN GOSUB 2000
18: 1150            TT = T2
19: 1160            GOTO 1210
20: 1170 '
21: 1180        GOSUB 2000: IF TQ(Q) < T1 THEN 1180
22: 1190        TT = T1
23: 1200 '
24: 1210        GOSUB 3000: NINZU = NINZU + 1: T2 = TQ(1)
25: 1220    WEND
26: 1230 '
27: 1240    PRINT "カウンター稼働率="; STIME / TT
28: 1250    PRINT
29: 1260    PRINT "平均待ち時間 ="; TW / NINZU
30: 1270 END
31: 1280 '
32: 2000 '---- 到着予定時刻
33: 2010 '
34: 2020    Q = Q + 1
35: 2030    TQ(Q) = TQ(Q - 1) - A * LOG(RND)
36: 2040    RETURN
37: 2050 '
38: 3000 '---- サービスの開始
39: 3010 '
40: 3020    TW = TW + (TT - TQ(1))
41: 3030    SE = -H * LOG(RND)
42: 3040    STIME = STIME + SE
43: 3050    T1 = TT + SE
44: 3060    FOR I = 0 TO Q
45: 3070        TQ(I) = TQ(I + 1)
46: 3080    NEXT I
47: 3090    Q = Q - 1
```

```
48: 3100    RETURN
49: 3110 '
50:
```

●リスト 2.2.3 QBへ変換後のソースファイル

```
 1:     '  save "henkaten.n88",a          '25-Sep-89
 2:     '
 3:       '----  待ち行列のシミュレーション（変化点方式）  ----
 4:         DIM TQ(500)
 5:       '
 6:         TEND = 50000!               'シミュレーション時間
 7:         A = 12                      '平均到着時間間隔
 8:         H = 10                      '平均サービス時間
 9:       '
10:         RANDOMIZE
11:         TW = 0: STIME = 0: TT = 0: TQ(0) = TT
12:         Q = 1: TQ(1) = TT - A * LOG(RND)
13:         T1 = TT - H * LOG(RND): T2 = TQ(1)
14:       '
15:         WHILE TT <= TEND
16:             IF T1 > T2 THEN 1180
17:                 IF Q <= 1 THEN GOSUB 2000
18:                 TT = T2
19:                 GOTO 1210
20:       '
21: 1180        GOSUB 2000: IF TQ(Q) < T1 THEN 1180
22:             TT = T1
23:       '
24: 1210        GOSUB 3000: NINZU = NINZU + 1: T2 = TQ(1)
25:         WEND
26:       '
27:         PRINT "カウンター稼働率="; STIME / TT
28:         PRINT
29:         PRINT "平均待ち時間 ="; TW / NINZU
30:       END
31:       '
32: 2000 '---- 到着予定時刻
33:       '
34:         Q = Q + 1
35:         TQ(Q) = TQ(Q - 1) - A * LOG(RND)
36:         RETURN
37:       '
38: 3000 '---- サービスの開始
39:       '
40:         TW = TW + (TT - TQ(1))
41:         SE = -H * LOG(RND)
42:         STIME = STIME + SE
43:         T1 = TT + SE
44:         FOR I = 0 TO Q
45:             TQ(I) = TQ(I + 1)
46:         NEXT I
47:         Q = Q - 1
48:         RETURN
49:       '
50:
```

2.3 パーソナル単語帳

2.3.1 パーソナル単語帳の概要

QBによる応用プログラム例として，単語帳のプログラムを作成してみます．

市販の手帳タイプの単語帳はよくできていますが，特定の分野の単語のみを利用するとか，学習用で，人によって必要とする語彙の範囲が異なるような場合には適しません．

ここでは，ユーザが単語登録を実行し，それをディスクファイルしておいて，使用するときに読み出したり，プリントさせたりする機能を持つようにします．

プログラミングの中心となるのは，ランダムファイルの使用と２分木検索です．２分木検索は，**1.7**で述べた形のものは，木がアンバランスになることがあるので，ここではバランス木あるいはAVL-木と呼ばれるものを用います．これについては，**2.5**で詳説します．

2.3.2 単語帳仕様

ユーザが登録した単語は，AVL-木構造を持つファイルとしてディスクにファイルされますが，登録や検索のたびにディスクアクセスをしていては，実行スピードが遅くなる可能性があります．そこで今回作った単語帳は，ディスクアクセスは最初と最後だけにして，途中ではファイルをメモリ上におくことにしました．これによって，快適なスピードでオペレーションができます．

記憶できる単語の語数は，ユーザが決めることができますが，プログラム例では100個にしてあります．登録単語数をいったん決めて辞書ファイルを作りはじめると，途中で変更がききませんから注意が必要です(最大約380個)．

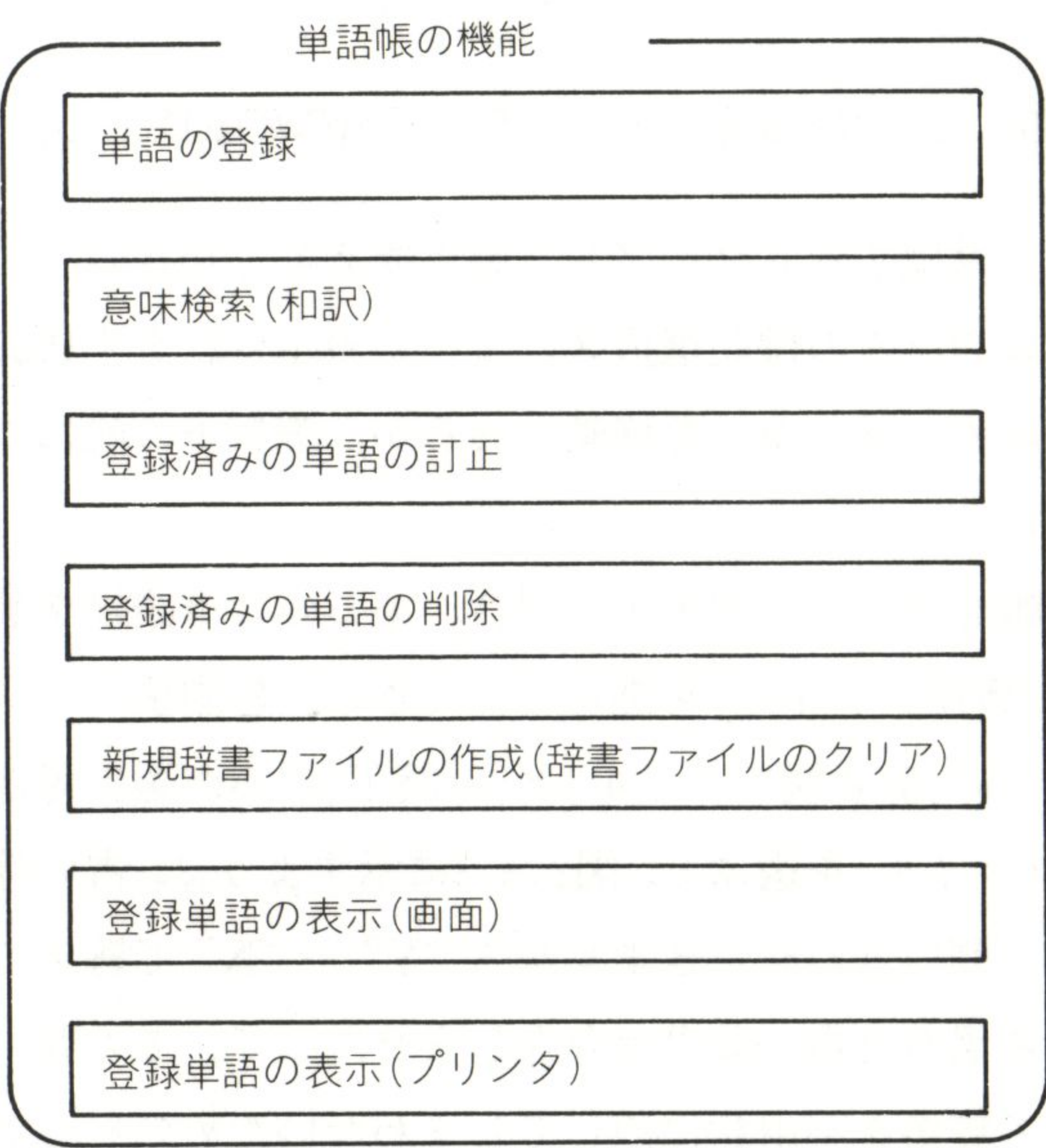
1語の大きさは，英単語のスペルが半角文字で20文字，訳が全角文字で35文字(半角で70文字)にしてあります．これらは一応変更が可能ですが，文字数を増やした場合，ディスプレイ画面のグラフィックス命令を大幅に書き直す必要があります．

この単語帳では，意味のほかにアクセントも登録することができます．アクセントが不明な場合は，この項目をパスすることも可能です．

図2.3.1にこの単語帳の機能を一覧します．

●図2.3.1 単語帳の機能

単語帳の機能

- 単語の登録
- 意味検索(和訳)
- 登録済みの単語の訂正
- 登録済みの単語の削除
- 新規辞書ファイルの作成(辞書ファイルのクリア)
- 登録単語の表示(画面)
- 登録単語の表示(プリンタ)

2.3.3 操作方法

(1) QBの環境に，メインモジュールであるDIC-MAIN. BASをロードします．MAKファイルにしたがって，サブモジュールであるDIC-AVL. BASとDIC-EDIT. BASが自動的にロードされます．また，実行ファイル(EXEファイル)が作成してある場合は，QB環境に入らずに，MS-DOSのコマンドラインより，次のように，

```
>DIC-MAIN
```

と入力すれば実行が始まります.

（2） プログラムを実行すると，まず辞書ファイルがオプーンされて，登録された単語がメモリに呼び込まれます．ここで辞書ファイルのドライブが一致しないか，指定したファイル“ENGWORD.DIC”が存在しない場合は，エラーメッセージを出してプログラムは停止します.

このプログラムでは，辞書ファイルはBドライブにおくように設定してあります．ドライブ名を変更したいときは，モジュールレベルコードのCONST文

```
CONST DicName$ = "B:ENGWORD.DIC"
```

の下線部を，希望するドライブ名に書き換えてください.

（3） 単語帳プログラムの機能選択メニューが表示されます（図2.3.2）．各機能を選択するにはカーソルキーを移動させるか，選択子の前についた数字を入力してください.

（4） この単語帳をはじめて使用する場合，あるいは単語帳を最初から作り直す場合には[5/新規ファイル]を選んでください．単語帳に1語も登録していないときにはこの[5/新規ファイル]と[8/終了]のみが選択可能です.

[5/新規ファイル]を選ぶと，図2.3.3に示すように，古い辞書を消去してよいかどうかの問い合わせがきますから“Y”か“N”で答えてください．はじめて使用する場合は古い辞書は存在しませんので，この問い合わせは，現在使用している辞書を不用意に消してしまわないためのものです．“Y”を選択すると辞書の初期化が行われたあと，機能選択メニュー画面に戻ります.

（5） 単語登録はメニューより[1/単語登録]を選択します．図2.3.4に示すような単語登録画面が現れます．この画面の基本構成は意味検索，訂正，削除の各機能の画面と同じになっています.

まず，単語のスペルを入力します．ここで，“.”（ピリオド）を入力すると[単語登録]機能は終了して，機能選択メニューに戻ることができます.

スペルの入力が終わったら，↵キーを押します．次はアクセントの入力に移ります．アクセントの入力はカーソルキーを用いて，カーソルを左右に移動させることによって行います．アクセント位置の確定は↵キーで行います.

●図2.3.2 単語帳機能選択メニュー画面

```
＊＊＊電子単語帳機能選択メニュー＊＊＊

                                        1990-09-04
                                         13:09:52
            1/単語登録
            2/意味検索
            3/訂正
            4/削除
            5/新規ファイル
            6/登録単語一覧
            7/印刷
            8/終了

選択したい機能をカーソル（↑・↓）で指定してください。
```

●図2.3.3「新規ファイル」機能選択画面

```
新規辞書ファイルを作成します.
古い辞書ファイルの内容は消去されます.
よろしいですか？(y/n)
```

●図2.3.4 単語登録画面

```
＊＊＊＊＊　単語登録　＊＊＊＊＊

                                        1990-09-04
                                         13:11:56

単語：

意味：

単語のスペルを入れてください。
（終了する時はピリオド（.）を入力）
```

カーソルは意味入力位置に移動すると同時に，単語のスペルのアクセント位置の英文字が赤色に変化します．単語のアクセント位置が不明で，入力をパスしたい場合は[ESC](エスケープ)キーを押します．単語のスペルはすべて白色のまま，意味入力に移ります．

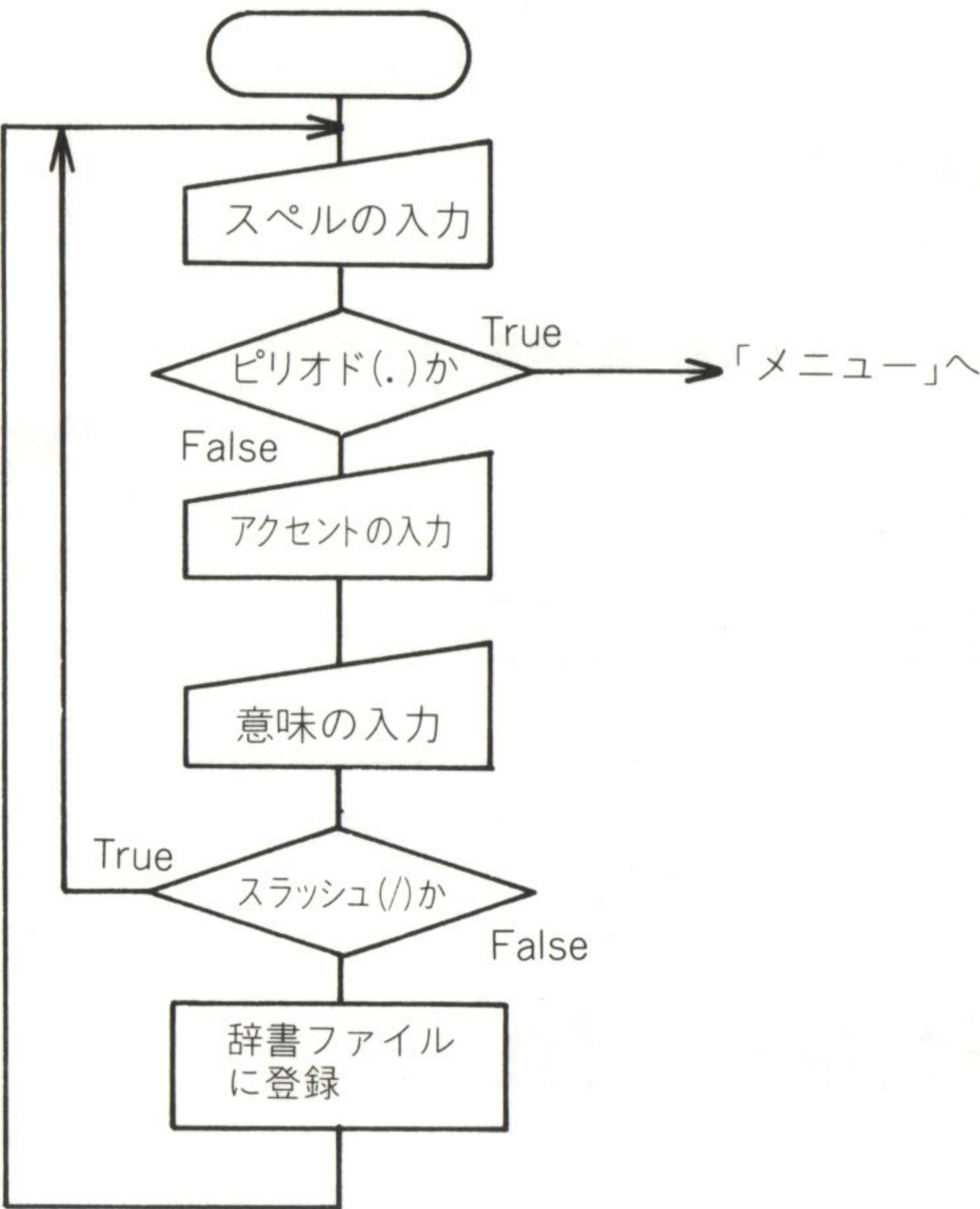
単語の意味は半角で70文字，全角で35文字入力できます．この時点で単語のミススペルなどに気づき，スペルの入力に戻りたいときは"/"(スラッシュ半角)を入力します．カーソルは入力済みの単語のスペルに戻り訂正ができます．訂正方法は，基本的には機能選択メニューから[3/訂正]を選択した場合と同じですので(7)を参照してください．

意味を入力し終わったら，↵キーを押してください．単語がメモリ上の辞書ファイルに登録されたあと，単語登録機能の最初の画面に戻り，連続して単語の登録ができます．単語登録を終了する場合は，単語のスペルを入力する代わりにピリオド(.)を入力します．

単語登録機能の手順の流れを図2.3.5に示します．

(6) 意味検索機能はメニューより[2/意味検索]を選択します．図2.3.6に意味検

●図2.3.5 単語登録手順

スペルの入力
ピリオド(.)か
True
「メニュー」へ
False
アクセントの入力
意味の入力
True
スラッシュ(/)か
False
辞書ファイル
に登録

索画面を示します．この機能は市販の電子辞書ではメインとなるものです．

この機能を選択すると，単語登録とほぼ同じ操作手順で登録された，単語の意味検索ができます．手順の流れを図2.3.7に示します．

意味を調べたい単語のスペルを入力して⏎キーを押せば，その単語のアク

●図2.3.6 意味検索画面

＊＊＊＊＊　意味検索　＊＊＊＊＊

1990-09-04
13:24:39

単語：

意味：

単語のスペルを入れてください。
（終了する時はピリオド（．）を入力）

●図2.3.7 意味検索手順

スペルの入力

ピリオド(.)か

True

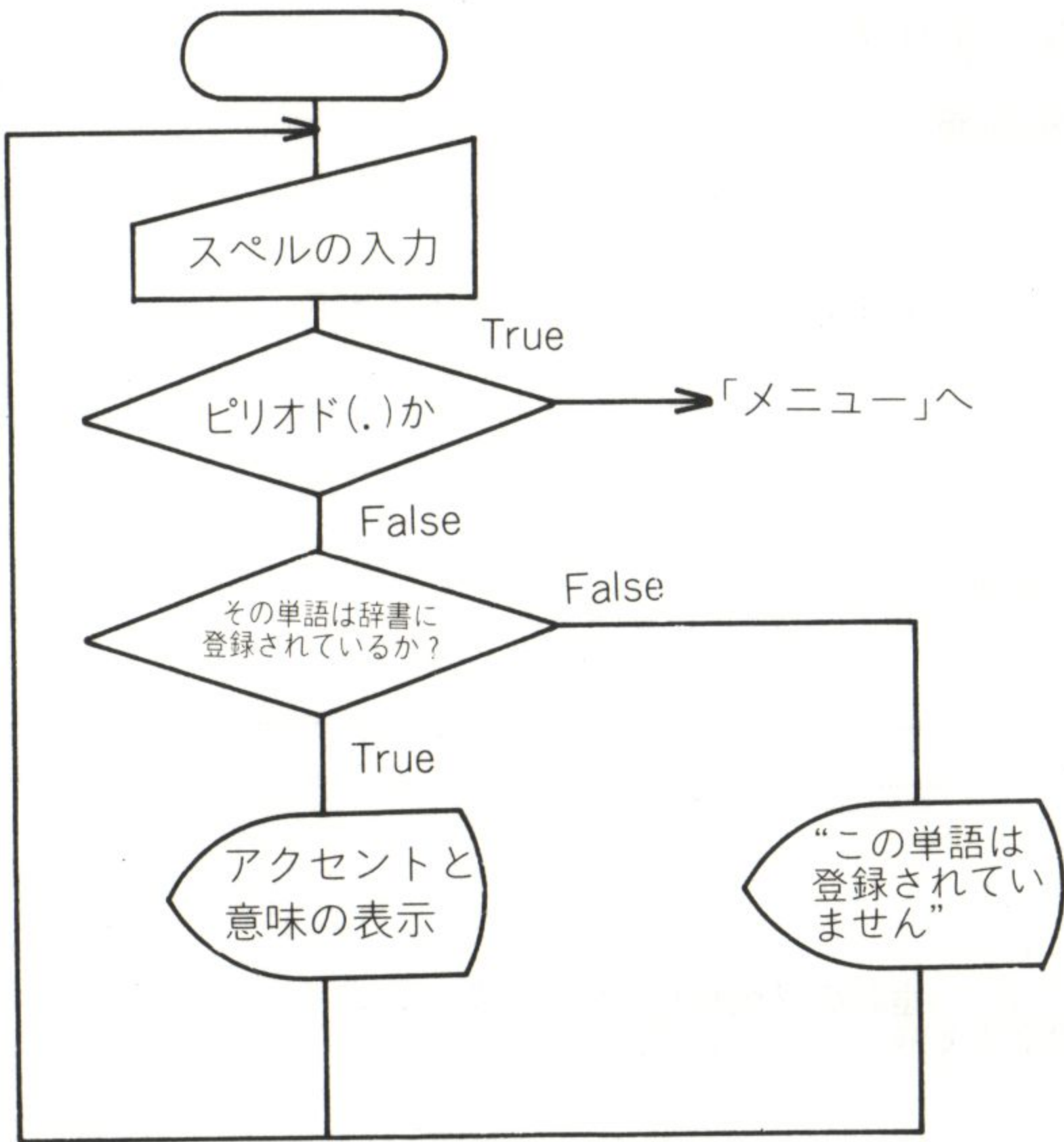
「メニュー」へ

False

その単語は辞書に登録されているか？

False

True

アクセントと意味の表示

“この単語は登録されていません”

セント位置の英字が赤色で示されるとともに，その意味が表示されます．

入力した単語が辞書ファイルに存在しない場合は,「この単語は登録されていません」という警告メッセージがピンク色で示されます．

アクセント位置が登録されていない単語を検索した場合には，当然アクセント位置の表示はありません．

意味検索もピリオド(.)を入力するまで連続して行うことができます．

(7) 訂正機能はメニューより[3/訂正]を選択します．図2.3.8に訂正画面を示します．手順の流れを図2.3.9に示します．

まず，訂正したい単語のスペルを入力します．その単語が辞書に登録されていないと「この単語は登録されていません」という警告メッセージを出して，スペルの再入力を要求します．登録されていればその単語のアクセント位置と意味を表示したあと，どの部分を訂正するか問い合わせるサブメニューが表示されますから，1～4までの数字で答えます(図2.3.10)．

[1：単語のスペル]を選択すると,カーソルが単語のスペルの先頭に現れて,エディットが可能になります．

[2：アクセント]を選択すると,アクセント位置入力が可能な状態になりますので，カーソルキーで位置を指定したあと，↵キーを押します．

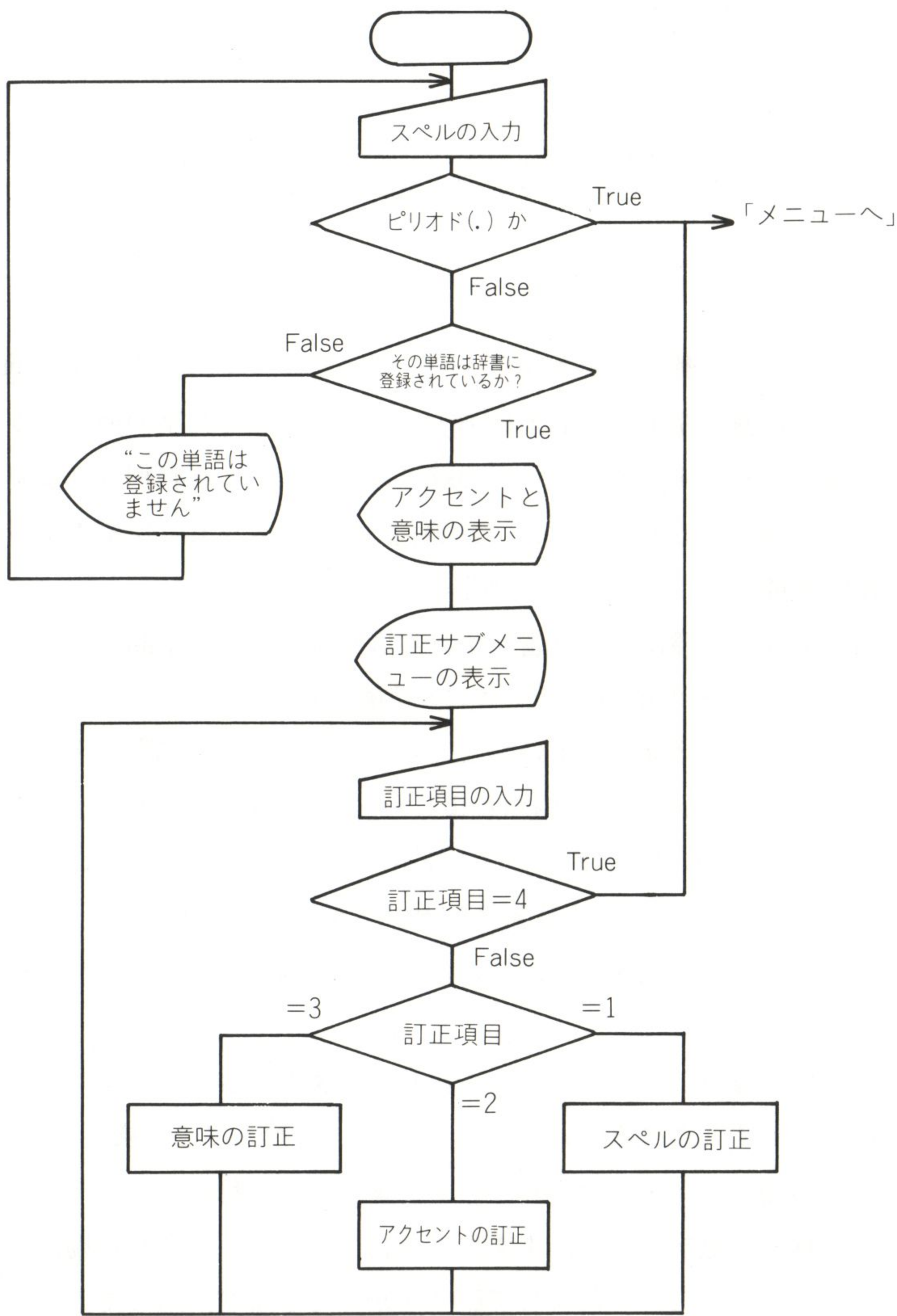
[3：意味]を選択すると，カーソルが単語の意味の先頭に現れて，エディット可能の状態になります．

●図2.3.8 訂正機能画面

```
                 *****    訂正    *****
                                          1990-09-04
                                           13:28:23

          単語：

   意味：

             訂正したい単語のスペルを入れてください。
             （終了する時はピリオド（.）を入力）
```

[4：訂正終了]を選択すると，訂正機能は終了して，機能選択メニュー画面に戻ります．1から3の機能は，4を入力するまで何度でも選択することができます．また，[1：単語のスペル]の訂正と[3：意味]の訂正を選択すると，それぞれ1行エディタ機能が働き，文字の挿入，1文字削除などが自由に行えます．

●図2.3.9 訂正手順

スペルの入力
ピリオド(.)か
True
「メニューへ」
False
その単語は辞書に登録されているか？
False
“この単語は登録されていません”
True
アクセントと意味の表示
訂正サブメニューの表示
訂正項目の入力
訂正項目＝4
True
False
訂正項目
＝3
＝1
＝2
意味の訂正
スペルの訂正
アクセントの訂正

●図2.3.10 訂正機能のサブメニュー画面

```
                    *****    訂正    *****
                                                    1990-09-04
                                                     13:31:01

          単語：indication

   意味：指示、印；<計器の>示度

          訂正したい項目の番号を入力してください.
          1:単語のスペル  2:アクセント  3:意味  4:訂正終了
          ? 3
```

カーソルの移動は矢印キー(←→)，カーソル位置の文字削除はDELキー，カーソル位置の左1文字消去はBSキーで行います.文字の挿入はカーソル位置に挿入されます．エディットの終了は↵キーにより判別されます．

訂正が終了すると，画面に「訂正完了」というメッセージが表示されます．

(8) 削除機能はメニューより[4/削除]を選択します．

訂正機能と同様，まず辞書ファイルより削除したい単語のスペルを入力します.すると,その単語のアクセント位置と意味が表示されたあと,図2.3.11に示すように，削除してよいかどうか確認がきます．“Y”と入力すると辞書ファイルより削除され，「削除しました」という確認メッセージが表示されます.削除したい単語が辞書ファイルの中に見つからないときは「この単語は登録されていません」という警告メッセージが表示されます．

(9) 新規ファイル作成機能は，メニューより[5/新規ファイル作成]を選択します．

この機能は，このプログラムをはじめて使うときと，辞書ファイルを新しく作りなおすときのみ使います.したがって,誤ってこの機能を選択すると,大切な辞書ファイルを消去してしまいますから注意が必要です．

この機能を選択すると，まず現在辞書ファイルが存在した場合，それが消去されてしまうことを警告して確認を求めてきます．誤ってこの項目を選ん

でしまった場合は“N”を入力すれば機能選択メニューに戻ることができます．

“Y”を入力した場合は，辞書ファイルの初期化が行われ，単語の登録が可能な状態になります．

(10) 登録単語一覧表示機能は，メニューより[6/登録単語一覧]を選択します．この機能は，辞書ファイルに登録されている単語を指定した範囲において，アルファベット順にディスプレイ上に表示させるものです．

この機能を選択したら，まず表示したい単語の範囲を指定します(図2.3.12)．範囲の上限と下限のアルファベットを入力すると，その範囲の頭文字を持つ登録単語のスペル，アクセント位置，意味がアルファベット順に表示されます．

アクセント位置の表示は，その部分の文字を赤色で表示することによって行っています．ディスプレイ上の表示だけではなく印刷したい場合は，次の[7/印刷]を選択してください．

●図2.3.11 削除機能における削除の確認画面

```
*****  削除  *****
                    1990-09-04
                    13:32:45

単語：benefit

意味：利益、利得；恩恵

削除してよろしいですか．(y/n)
```

●図2.3.12 表示したい単語の範囲指定画面

```
表示したい単語の範囲を指定します．
範囲の最初の頭文字を入力してください．？a
      最後の頭文字を入力してください．？c
```

(11)　印刷機能はメニューより[7/印刷]を選択します．この機能は，辞書ファイルに登録されている単語を，指定した範囲においてアルファベット順にプリンタに印刷させるものです．

まず，プリンタを準備するようにメッセージがきますから，プリンタにA4用紙をセットして，“Y”を入力してください．

範囲の指定，および表示のスタイルは[6/登録単語一覧]とほぼ同じです．ただし，アクセントの位置表示はアンダーラインによって行っています．

図2.3.13に印刷例を示します．

(12)　このプログラムを終了したい場合には[8/終了]を選択します．**2.3.2**の単語帳仕様の項でも述べましたが，辞書ファイルは，プログラム実行の最初に，いったんすべてフロッピィディスクからメモリ上に呼び出しておいて，辞書ファイルへのアクセスはこのメモリ上のファイルに対して行われます．そこでプログラム終了時には，メモリ上の辞書ファイルをフロッピィディスクに書き戻してやる必要があります．終了操作は，この作業を行ってからプログ

●図2.3.13 印刷例

＊＊＊　登録単語リスト　＊＊＊

a	：不定冠詞
arrow	：矢
beginner	：初心者，およびビギナー
bellow	：大声でなく、吠える
belong	：属する(to)，ある、いる
benefit	：利益、利得；恩恵
blind	：盲の、目のよく見えない
blush	：赤面、顔を赤らめる
book	：本
border	：縁、端；国境
both	：双方の、両方の；both...andで、..も..も、のみならず
breathe	：呼吸する、生きている；息切れさせる；ささやく；..again ほっとする

ラムを終了させます．したがって，終了操作をせずにパソコンの電源を切ってしまった場合などは，その時に行った単語登録や訂正などがすべて無効になってしまいます．

2.3.4 モジュール

このプログラムは，3つのモジュール(ひとつのメインモジュールと2つのサブモジュール)によってプログラムが構成されています．

メインモジュールはDIC-MAIN. BASで，プログラム全体の流れをコントロールするプロシージャ，および単語帳の各機能を処理するプロシージャからなっています．

サブモジュールのひとつ目は，AVL-木構造を持つ辞書ファイル関係のプロシージャから構成されたDIC-AVL. BAS，もうひとつのサブモジュールは，単語帳でスペルや意味の入力および修正に使用される，1行エディタを構成するプロシージャからなるDIC-EDIT. BASです．

辞書ファイルのデータは，Node型と呼ばれるレコード型の配列変数xを用意し，その各要素に格納されることになります．配列要素の数は辞書ファイルで扱える単語の数にあたり，定数ListSizeで与えます．Node型は次のようなTYPE文を使って，メインモジュールとサブモジュールのモジュールレベルコードにおいて，宣言しておきます．

```
TYPE Node
    inf AS STRING * InfSize
    acc AS INTEGER
    synop AS STRING * SynopSize
    left AS INTEGER
    right AS INTEGER
    weight AS INTEGER
END TYPE
```

ここで定数InfSizeは単語の長さ(文字数)，定数SynopSizeは単語の意味を記述

する文字数を指す定数で，TYPE宣言より前に次のようにCONST宣言しておきます．

```
CONST InfSize = 20
CONST SynopSize = 70
CONST ListSize = 100
```

定数ListSizeは辞書ファイルの登録単語数を指す定数です．

ところで，この配列変数xはメインモジュールだけではなく，サブモジュールからもアクセスされますから，モジュール間で共有される必要があります．またそれぞれのモジュール内ではグローバルである必要があります．そこで，この配列変数xについては，両方のモジュールのモジュールレベルコードで次のようにCOMMON SHARED宣言をしておかなければなりません．この場合，配列要素の数はDIM文で宣言しますから省略してかまいません．

```
DIM x(ListSize)  AS Node
COMMON SHARED x( ) AS Node
```

2.3.5 主要なプロシージャについての説明

メインモジュールのリストを**リスト2.3.1**に示します．サブモジュールのリストは**2.4**(188ページ)および**2.5**(209ページ)に示します．

以下に，メインモジュールの主要なプロシージャについて，説明します．

2.3.5.1 モジュールレベルコード

メインモジュールのモジュールレベルコードでは，メインモジュールからコールするSUB，あるいはFUNCTIONプロシージャの宣言をするDECLARE文，定数の宣言，ユーザ定義変数の宣言とグローバル変数の宣言を行っています．

また，タイマートラッピングの宣言とエラートラッピングの宣言，およびそれぞれの処理ルーチンも記述されています．ただし，これらの処理ルーチンは，処理の流れに影響を受けないようにEND文の後方に位置していることに注意してく

ださい.

READ文でメニュー項目を文字列配列MenuTitle$に読み込ませますが, DATA文もモジュールレベルコードに置く必要があります.

SUBプロシージャMenuをコールして, モジュールレベルコードの処理は終わります.

2.3.5.2 SUBプロシージャ Menu

このプロシージャがモジュールレベルコードからコールされると, まずフロッピィディスク上のランダム辞書ファイルENGWORD. DICが, Node型のグローバル配列変数x(ListSize)に読み込まれます. これは, **2.3.2**で書いたように, 辞書ファイルの書き換え, 検索などで, フロッピィディスク上のランダムファイルが頻繁にアクセスされると, 実行が遅くなり快適な使用感が得られなくなります. そこで, 処理の最初に, フロッピィディスク上の辞書ファイルをあらかじめメモリ上の配列に移しておきます.

次に, パーソナル単語帳の各機能を呼び出すメニュー処理に移ります. ここで, タイマートラッピングを開始するためにTIMER ONステートメントを書いておきます.

FUNCTIONプロシージャDispMenuを使用して, 単語帳の機能一覧メニューを表示させて処理を選択させます. カーソルあるいはショートカットキーを使って機能を選択すると, この関数は機能に対応した数値を返します.

p%が0でない, つまりこの単語帳を初めて使うとき以外は, SELECT CASE文によって, 各機能ごとのプロシージャに実行が振り分けられます. p%が0のときは, メニューの[5/新規ファイル作成]を選択してからでないと, ほかの機能を選択できないようになっています([8/終了]は別).

もし, ほかの機能を選択した場合は, 警告メッセージを表示するようになっています.

さて, メニューの1~7の機能のうち, 画面に時刻を表示すると邪魔になってしまう5,6,7の場合は, タイマートラッピングを中止するために, 各プロシージャをコールする前にTIMER OFFステートメントを置き, プロシージャから戻ったら, TIMER ONステートメントを置いて, トラッピングを再開するようにしています.

●リスト 2.3.1 メインモジュールDIC-MAIN・BASの全リスト

```
 1: DECLARE SUB GetAccPos (AccPosGet!, WordSpell$)
 2: DECLARE SUB DeleteStr (s1$, start%, Num%)
 3: DECLARE SUB InsertStr (s2$, s1$, PosNum%)
 4: DECLARE FUNCTION IsKanji! (C$)
 5: DECLARE SUB ReadLine (s1$, BufLen!, x0, y0)
 6: DECLARE SUB DispFile (p%)
 7: DECLARE SUB DispTree (p%, depth!, u$, l$)
 8: DECLARE SUB Delete (t$, p%, f%)
 9: DECLARE SUB Insert (a$, B%, C$, p%)
10: DECLARE SUB PrintTree (p%, depth!, u$, l$)
11: DECLARE SUB PrtFile (p%)
12: DECLARE SUB Correct (p%)
13: DECLARE SUB DelWord (p%)
14: DECLARE SUB Translate (p%)
15: DECLARE SUB InitArray ()
16: DECLARE SUB Entry (p%)
17: DECLARE SUB NewFile (p%)
18: DECLARE FUNCTION DispMenu! (MenuNum!, MenuTitle$())
19: DECLARE SUB Menu (MenuNum, MenuTitle$())
20: DECLARE SUB InitFreeList ()
21: DECLARE FUNCTION Search! (Item$, p%)
22:
23: CONST InfSize = 20
24: CONST SynopSize = 70
25: CONST ListSize = 100
26: CONST False = 0, True = NOT False
27:
28: CONST DicName$ = "B:ENGWORD.DIC"
29:
30: CONST MenuNum = 8
31:
32: CONST JobTitleRow = 3
33: CONST JobTitleCol = 20
34: CONST WordRow = 10
35: CONST WordCol = 20
36: CONST MeanRow = 14
37: CONST MeanCol = 8
38: CONST CommandRow = 20
39: CONST CommandCol = 10
40: CONST ClockRow = 5
41: CONST ClockCol = 60
42:
43:
44: TYPE Node
45:   inf AS STRING * InfSize
46:   acc AS INTEGER
47:   synop AS STRING * SynopSize
48:   left AS INTEGER
49:   right AS INTEGER
50:   weight AS INTEGER
51: END TYPE
52:
53: DIM x(ListSize)  AS Node
54: DIM MenuTitle$(10)
55:
56: COMMON SHARED x()  AS Node
57:
58:     ON TIMER(1) GOSUB DispTime
59:
60:     ON ERROR GOTO ErrExit
61:
62:     CLS 0
63:     SCREEN 0
64:
65:     FOR I = 1 TO MenuNum
66:         READ MenuTitle$(I)
```

```
 67:     NEXT I
 68:
 69:     DATA 1/単語登録，2/意味検索，3/訂正，4/削除
 70:     DATA 5/新規ファイル，6/登録単語一覧，7/印刷，8/終了
 71:
 72:     CALL Menu(MenuNum, MenuTitle$())
 73:     CLS
 74:
 75:     END
 76:
 77: ErrExit:
 78:
 79:     PRINT "エラーが発生しました．"
 80:     PRINT "エラー番号＝"; ERR
 81:
 82:     SELECT CASE ERR
 83:         CASE 71
 84:           PRINT "指定したドライブに辞書がありません"
 85:           PRINT "辞書ファイルの入ったフロッピィディスクをセットして"
 86:           PRINT "なにかキーを押してください．"
 87:           DO
 88:           LOOP UNTIL (INKEY$ <> "")
 89:         CASE ELSE
 90:             CLOSE
 91:             END
 92:
 93:     END SELECT
 94:
 95:
 96: RESUME
 97:
 98:
 99: DispTime:
100:
101:     XPos = POS(0)
102:     YPos = CSRLIN
103:
104:     LOCATE ClockRow, ClockCol
105:     PRINT DATE$
106:     LOCATE ClockRow + 1, ClockCol + 1
107:     PRINT TIME$
108:
109:     LOCATE YPos, XPos
110:
111: RETURN
112:
113: SUB Correct (p%)
114:
115: DIM OneWord AS Node
116: DIM Spell(1 TO InfSize) AS STRING * 1
117:
118: DO
119:     CLS
120:
121:     LOCATE JobTitleRow, JobTitleCol
122:     PRINT "＊＊＊＊＊     訂正      ＊＊＊＊＊"
123:
124:     LOCATE WordRow, WordCol - 6
125:     PRINT "単語："
126:     LINE (100, 140)-(314, 163), , B
127:
128:     LOCATE MeanRow, MeanCol - 6
129:     PRINT "意味："
130:     LOCATE MeanRow, MeanCol
131:     LINE (1, 204)-(620, 227), , B
132:
133:     LOCATE CommandRow, CommandCol
134:     PRINT "訂正したい単語のスペルを入れてください。"
135:     LOCATE CommandRow + 1, CommandCol
```

```
136:     PRINT "（終了する時はピリオド（．）を入力）" + SPACE$(40)
137:
138:     s$ = ""
139:     CALL ReadLine(s$, InfSize, WordCol, WordRow)
140:     OneWord.inf = s$
141:     IF INSTR(OneWord.inf, ".") THEN EXIT SUB
142:
143:     LOCATE WordRow, WordCol
144:
145:     LengthWord = LEN(RTRIM$(OneWord.inf))
146:     IF LengthWord > InfSize THEN LengthWord = InfSize
147:
148:     FOR I = 1 TO LengthWord
149:         Spell(I) = MID$(OneWord.inf, I, 1)
150:         PRINT Spell(I);
151:     NEXT I
152:
153:     s$ = LEFT$(OneWord.inf, 20)
154:     where = Search(s$, p%)
155:     IF where <> 0 THEN
156:         AccentPos = x(where).acc
157:         OneWord.acc = AccentPos
158:         OldAccentPos = AccentPos
159:         OneWord.synop = x(where).synop
160:         LastAccentPos = AccentPos
161:
162:
163:         IF AccentPos <> 0 THEN
164:             LOCATE WordRow, WordCol + AccentPos - 1
165:             COLOR 4
166:             PRINT Spell(AccentPos)
167:             COLOR 7
168:         END IF
169:
170:
171:         LOCATE MeanRow, MeanCol
172:         PRINT OneWord.synop
173:
174:         DO
175:             LOCATE CommandRow, CommandCol
176:             PRINT "訂正したい項目の番号を入力してください．"
177:             LOCATE CommandRow + 1, CommandCol
178:             PRINT "1:単語のスペル  2:アクセント  3:意味  4:訂正終了"
179:             LOCATE CommandRow + 2, CommandCol
180:             PRINT SPACE$(40)
181:             LOCATE CommandRow + 2, CommandCol
182:             INPUT Item$
183:             a$ = LEFT$(Item$, 1)
184:
185:             LOCATE CommandRow - 2, CommandCol
186:             PRINT SPACE$(40)
187:
188:             SELECT CASE a$
189:
190:                 CASE "1"
191:
192:                     s$ = RTRIM$(OneWord.inf)
193:                     CALL ReadLine(s$, InfSize, WordCol, WordRow)
194:                     OneWord.inf = s$
195:
196:                     LengthWord = LEN(OneWord.inf)
197:
198:                     LOCATE WordRow, WordCol
199:                     PRINT SPACE$(LengthWord)
200:
201:                     LOCATE WordRow, WordCol
202:                     FOR I = 1 TO LengthWord
203:                         Spell(I) = MID$(OneWord.inf, I, 1)
204:                         PRINT Spell(I);
```

```
205:                NEXT I
206:
207:                IF AccentPos > LengthWord THEN AccentPos = LengthWord
208:                OneWord.acc = AccentPos
209:                OldAccentPos = AccentPos
210:
211:                IF AccentPos <> 0 THEN
212:                    LOCATE WordRow, WordCol + AccentPos - 1
213:                    COLOR 4
214:                    PRINT Spell(AccentPos)
215:                    COLOR 7
216:                END IF
217:
218:            CASE "2"
219:
220:                WordSpell$ = RTRIM$(OneWord.inf)
221:                CALL GetAccPos(AccentPos, WordSpell$)
222:
223:                OneWord.acc = AccentPos
224:
225:
226:                IF AccentPos <> 0 THEN
227:
228:                    IF LastAccentPos <> 0 THEN
229:                        LOCATE WordRow, WordCol + LastAccentPos - 1
230:                        COLOR 7
231:                        PRINT Spell(LastAccentPos)
232:                    END IF
233:
234:                    LOCATE WordRow, WordCol + AccentPos - 1
235:                    COLOR 4
236:                    PRINT Spell(AccentPos)
237:                    COLOR 7
238:                    LastAccentPos = AccentPos
239:
240:                END IF
241:
242:
243:            CASE "3"
244:                s$ = RTRIM$(OneWord.synop)
245:                CALL ReadLine(s$, SynopSize, MeanCol, MeanRow)
246:                OneWord.synop = s$
247:
248:            CASE "4"
249:                EXIT DO
250:        END SELECT
251:
252:        COLOR 5
253:        LOCATE CommandRow - 2, CommandCol
254:        PRINT "＊＊＊  訂正完了  ＊＊＊"
255:        COLOR 7
256:    LOOP
257:
258:    CALL Insert(OneWord.inf, OneWord.acc, OneWord.synop, p%)
259:
260:    IF a$ = "4" THEN EXIT SUB
261:
262: ELSE
263:    LOCATE MeanRow, MeanCol
264:    COLOR 5
265:    PRINT "この単語は登録されていません．"
266:    COLOR 7
267: END IF
268:
269: LOCATE CommandRow, CommandCol
270: PRINT "何かキーを押してください．" + SPACE$(40)
271: LOCATE CommandRow + 1, CommandCol
272: PRINT SPACE$(40)
273: DO
```

```
274:     LOOP UNTIL INKEY$ <> ""
275:
276: LOOP
277:
278: END SUB
279:
280: SUB DelWord (p%)
281:
282: DIM OneWord AS Node
283: DIM Spell(1 TO InfSize) AS STRING * 1
284:
285: DO
286:     CLS
287:
288:     LOCATE JobTitleRow, JobTitleCol
289:     PRINT "＊ ＊ ＊ ＊ ＊     削 除     ＊ ＊ ＊ ＊ ＊"
290:
291:     LOCATE WordRow, WordCol - 6
292:     PRINT "単 語 ： "
293:     LINE (100, 140)-(314, 163), , B
294:
295:     LOCATE MeanRow, MeanCol - 6
296:     PRINT "意 味 ： "
297:     LOCATE MeanRow, MeanCol
298:     LINE (1, 204)-(620, 227), , B
299:
300:     LOCATE CommandRow, CommandCol
301:     PRINT "削 除 し た い 単 語 の ス ペ ル を 入 れ て く だ さ い 。"
302:     LOCATE CommandRow + 1, CommandCol
303:     PRINT "（ 終 了 す る 時 は ピ リ オ ド （ ． ） を 入 力 ） " + SPACE$(40)
304:
305:     s$ = ""
306:     CALL ReadLine(s$, InfSize, WordCol, WordRow)
307:     OneWord.inf = s$
308:     IF INSTR(OneWord.inf, ".") THEN EXIT SUB
309:
310:     LOCATE WordRow, WordCol
311:
312:     LengthWord = LEN(RTRIM$(OneWord.inf))
313:     IF LengthWord > InfSize THEN LengthWord = InfSize
314:
315:     FOR I = 1 TO LengthWord
316:         Spell(I) = MID$(OneWord.inf, I, 1)
317:         PRINT Spell(I);
318:     NEXT I
319:
320:     s$ = LEFT$(OneWord.inf, 20)
321:
322:     where = Search(s$, p%)
323:     IF where <> 0 THEN
324:         AccentPos = x(where).acc
325:
326:         IF AccentPos <> 0 THEN
327:             LOCATE WordRow, WordCol + AccentPos - 1
328:             COLOR 4
329:             PRINT Spell(AccentPos)
330:             COLOR 7
331:         END IF
332:
333:         LOCATE MeanRow, MeanCol
334:         PRINT x(where).synop
335:         LOCATE CommandRow, CommandCol, 0
336:         PRINT "削 除 し て よ ろ し い で す か ． (y/n)" + SPACE$(20)
337:         LOCATE CommandRow + 1, CommandCol
338:         PRINT SPACE$(40)
339:
340:         DO
341:             a$ = INKEY$
342:         LOOP UNTIL (UCASE$(a$) = "Y" OR UCASE$(a$) = "N")
```

```
343:
344:        LOCATE , , 1
345:
346:        IF UCASE$(a$) = "Y" THEN
347:
348:            CALL Delete(s$, p%, f%)
349:
350:            LOCATE MeanRow, MeanCol
351:            COLOR 5
352:            PRINT "削除しました. " + SPACE$(70)
353:            COLOR 7
354:        END IF
355:    ELSE
356:        LOCATE MeanRow, MeanCol
357:        COLOR 5
358:        PRINT "この単語は登録されていません. "
359:        COLOR 7
360:    END IF
361:
362:    LOCATE CommandRow, CommandCol
363:    PRINT "何かキーを押してください. " + SPACE$(40)
364:    LOCATE CommandRow + 1, CommandCol
365:    PRINT SPACE$(40)
366:
367:    DO
368:    LOOP UNTIL INKEY$ <> ""
369: LOOP
370:
371: END SUB
372:
373: SUB DispFile (p%)
374:
375:    CLS 0
376:
377:    PRINT "表示したい単語の範囲を指定します. "
378:
379:    DO
380:        INPUT "範囲の最初の頭文字を入力してください. "; u$
381:        INPUT "      最後の頭文字を入力してください. "; l$
382:        u$ = UCASE$(LEFT$(u$, 1))
383:        l$ = UCASE$(LEFT$(l$, 1))
384:        IF u$ > l$ THEN
385:            PRINT "範囲指定がおかしいです. "
386:        ELSE
387:            EXIT DO
388:        END IF
389:    LOOP
390:
391:    CLS 0
392:    CALL DispTree(p%, 0, u$, l$)
393:
394:    PRINT "何かキーをおしてください. "
395:    DO
396:    LOOP UNTIL INKEY$ <> ""
397:
398: END SUB
399:
400: FUNCTION DispMenu (Num, Title$())
401:
402:    CLS
403:
404:    LOCATE JobTitleRow, JobTitleCol
405:    PRINT "＊＊＊電子単語帳機能選択メニュー＊＊＊"
406:
407:    LOCATE CommandRow, CommandCol
408:    PRINT "選択したい機能をカーソル（↑・↓）で指定してください。"
409:
410:
411:    row = 7
```

```
412:    col = 30
413:
414:    I = 1
415:    LOCATE row, col
416:    COLOR 15
417:    PRINT USING "&                  &"; Title$(I)
418:
419:    COLOR 7
420:    FOR I = 2 TO Num
421:        LOCATE row + I - 1, col
422:        PRINT USING "&                  &"; Title$(I)
423:    NEXT I
424:
425:    filenum = 1
426:
427:    DO
428:        DO
429:            a$ = INKEY$
430:        LOOP UNTIL ((VAL(a$) >= 1 AND VAL(a$) <= 8)) OR (a$ = CHR$(0, &H48) OR a$ =
        CHR$(0, &H50) OR a$ = CHR$(13))
431:
432:        IF VAL(a$) >= 1 AND VAL(a$) <= 8 THEN
433:            DispMenu = VAL(a$)
434:            EXIT FUNCTION
435:        END IF
436:
437:
438:        DO WHILE (a$ = CHR$(0, &H48) OR a$ = CHR$(0, &H50))
439:
440:            SELECT CASE a$
441:                CASE CHR$(0, &H48)
442:                    nextfnum = filenum - 1
443:                    IF nextfnum < 1 THEN
444:                        nextfnum = Num
445:                    END IF
446:                CASE CHR$(0, &H50)
447:                    nextfnum = filenum + 1
448:                    IF nextfnum > Num THEN
449:                        nextfnum = 1
450:                    END IF
451:            END SELECT
452:
453:            IF (nextfnum >= 1 AND nextfnum <= Num) THEN
454:                COLOR 7
455:                LOCATE row + filenum - 1, col
456:                PRINT USING "&                  &"; Title$(filenum)
457:
458:                COLOR 15
459:                LOCATE row + nextfnum - 1, col
460:                PRINT USING "&                  &"; Title$(nextfnum)
461:
462:                filenum = nextfnum
463:
464:                COLOR 7
465:
466:            END IF
467:
468:            DO
469:                a$ = INKEY$
470:            LOOP WHILE a$ = ""
471:        LOOP
472:
473:    LOOP UNTIL (a$ = CHR$(13))
474:
475:    COLOR 7
476:
477:    DispMenu = filenum
478:
479: END FUNCTION
```

```
480:
481: SUB Entry (p%)
482:
483: DIM OneWord AS Node
484: DIM Spell(1 TO InfSize) AS STRING * 1
485: DIM WordSynop AS STRING * SynopSize
486:
487: DO
488:
489: CLS
490:
491: s$ = ""
492:
493: InpSpell:
494:
495: LOCATE JobTitleRow, JobTitleCol
496: PRINT "＊＊＊＊＊　単語登録　＊＊＊＊＊"
497:
498: LOCATE WordRow, WordCol - 6
499: PRINT "単語："
500: LINE (100, 140)-(314, 163), , B
501:
502: LOCATE MeanRow, MeanCol - 6
503: PRINT "意味："
504: LOCATE MeanRow, MeanCol
505: LINE (1, 204)-(620, 227), , B
506:
507: LOCATE CommandRow, CommandCol
508: PRINT "単語のスペルを入れてください。"
509: LOCATE CommandRow + 1, CommandCol
510: PRINT "（終了する時はピリオド（．）を入力）" + SPACE$(40)
511:
512: CALL ReadLine(s$, InfSize, WordCol, WordRow)
513: OneWord.inf = s$
514: IF INSTR(OneWord.inf, ".") THEN EXIT SUB
515:
516: s$ = LEFT$(OneWord.inf, 20)
517: where = Search(s$, p%)
518:
519: IF where <> 0 THEN
520:     COLOR 5
521:     LOCATE CommandRow, CommandCol, 0
522:     PRINT "この単語はすでに登録されています．" + SPACE$(40)
523:     LOCATE CommandRow + 1, CommandCol
524:     PRINT "上書きしますか．(y/n)" + SPACE$(40)
525:     COLOR 7
526:
527:     DO
528:         a$ = INKEY$
529:     LOOP UNTIL (UCASE$(a$) = "Y" OR UCASE$(a$) = "N")
530:
531:     LOCATE CommandRow, CommandCol
532:     PRINT SPACE$(40)
533:     LOCATE CommandRow + 1, CommandCol
534:     PRINT SPACE$(40)
535:
536:     LOCATE , , 1
537:
538:     IF UCASE$(a$) = "N" THEN
539:         EXIT SUB
540:     END IF
541: END IF
542:
543: LengthWord = LEN(RTRIM$(OneWord.inf))
544: LOCATE WordRow, WordCol
545: FOR I = 1 TO LengthWord
546:     Spell(I) = MID$(OneWord.inf, I, 1)
547:     PRINT Spell(I);
548: NEXT I
```

```
549:
550: WordSpell$ = RTRIM$(OneWord.inf)
551: CALL GetAccPos(AccentPos, WordSpell$)
552:
553: IF AccentPos <> 0 THEN
554:     COLOR 4
555:     LOCATE WordRow, WordCol + AccentPos - 1
556:     PRINT Spell(AccentPos)
557:     COLOR 7
558: END IF
559:
560: DO
561:     LOCATE CommandRow, CommandCol, 1
562:     PRINT "単語の意味を入れてください。" + SPACE$(40)
563:     LOCATE CommandRow + 1, CommandCol
564:     PRINT "スペルの入力に戻りたいときはスラッシュ（／）を入力．"
565:
566:     s$ = ""
567:     CALL ReadLine(s$, SynopSize, MeanCol, MeanRow)
568:     WordSynop = LEFT$(s$, SynopSize)
569:
570:     IF INSTR(WordSynop, "/") THEN
571:         LOCATE MeanRow, MeanCol, 0
572:         PRINT SPACE$(SynopSize)
573:         s$ = RTRIM$(OneWord.inf)
574:         GOTO InpSpell
575:     END IF
576:
577: LOOP UNTIL WordSynop <> SPACE$(SynopSize)
578:
579: OneWord.acc = AccentPos
580: OneWord.synop = WordSynop
581:
582: CALL Insert(OneWord.inf, OneWord.acc, OneWord.synop, p%)
583:
584: LOOP
585:
586: END SUB
587:
588: SUB GetAccPos (AccPosGet, WordSpell$)
589:
590: DIM Spell(1 TO InfSize) AS STRING * 1
591:
592: LengthWord = LEN(WordSpell$)
593: FOR I = 1 TO LengthWord
594:     Spell(I) = MID$(WordSpell$, I, 1)
595: NEXT I
596:
597:
598: LOCATE WordRow, WordCol
599: AccPosGet = 1
600: COLOR 15
601: PRINT Spell(AccPosGet)
602: COLOR 7
603:
604: LOCATE CommandRow, CommandCol, 0
605: PRINT "単語のアクセントをカーソル（←・→）" + SPACE$(20)
606: LOCATE CommandRow + 1, CommandCol
607: PRINT "で指定してください。     (pass->ESC)" + SPACE$(20)
608: LOCATE CommandRow + 2, CommandCol
609: PRINT SPACE$(40)
610:
611: DO
612:     DO
613:         a$ = INKEY$
614:     LOOP WHILE a$ = ""
615:
616:     DO WHILE (a$ = CHR$(0, &H4B) OR a$ = CHR$(0, &H4D))
617:
```

```
618:        SELECT CASE a$
619:            CASE CHR$(0, &H4B)
620:                NewAccentPos = AccPosGet - 1
621:            CASE CHR$(0, &H4D)
622:                NewAccentPos = AccPosGet + 1
623:        END SELECT
624:
625:        IF (NewAccentPos >= 1 AND NewAccentPos <= LengthWord) THEN
626:            COLOR 7
627:            LOCATE WordRow, WordCol + AccPosGet - 1
628:            PRINT Spell(AccPosGet)
629:
630:
631:            COLOR 15
632:            LOCATE WordRow, WordCol + NewAccentPos - 1
633:            PRINT Spell(NewAccentPos)
634:            AccPosGet = NewAccentPos
635:            COLOR 7
636:
637:        END IF
638:
639:        DO
640:            a$ = INKEY$
641:        LOOP WHILE a$ = ""
642:    LOOP
643:    IF a$ = CHR$(&H1B) THEN
644:        AccPosGet = 0
645:        LOCATE WordRow, WordCol
646:        COLOR 7
647:        FOR I = 1 TO LengthWord
648:            PRINT Spell(I);
649:        NEXT I
650:    END IF
651:
652: LOOP UNTIL (a$ = CHR$(13) OR a$ = CHR$(&H1B))
653:
654: END SUB
655:
656: SUB InitArray
657:
658: FOR I = 0 TO ListSize
659:     x(I).inf = ""
660:     x(I).acc = 0
661:     x(I).synop = ""
662:     x(I).left = 0
663:     x(I).right = 0
664:     x(I).weight = 0
665: NEXT I
666:
667: END SUB
668:
669: SUB Menu (MenuNum, MenuTitle$())
670:
671: DIM xdata AS Node
672:
673: OPEN DicName$ FOR RANDOM AS #1 LEN = LEN(xdata)
674:
675:     FOR j = 0 TO ListSize
676:         GET #1, j + 1, x(j)
677:     NEXT j
678:
679: CLOSE #1
680:
681: TIMER ON
682:
683: p% = x(0).left
684:
685: DO
686:     LOCATE , , 0
```

```
687:     SelectedJob = DispMenu(MenuNum, MenuTitle$())
688:     LOCATE , , 1
689:
690:     IF p% <> 0 THEN
691:
692:         SELECT CASE SelectedJob
693:             CASE 1    '単語登録
694:                 CALL Entry(p%)
695:             CASE 2    '意味検索
696:                 CALL Translate(p%)
697:             CASE 3    '訂正
698:                 CALL Correct(p%)
699:             CASE 4    '削除
700:                 CALL DelWord(p%)
701:             CASE 5    '新規ファイル作成
702:                 TIMER OFF
703:                 CALL NewFile(p%)
704:                 TIMER ON
705:             CASE 6    '登録単語一覧表示
706:                 TIMER OFF
707:                 CALL DispFile(p%)
708:                 TIMER ON
709:             CASE 7    '印刷
710:                 TIMER OFF
711:                 CALL PrtFile(p%)
712:                 TIMER ON
713:             CASE 8    '終了
714:                 EXIT DO
715:         END SELECT
716:     ELSE
717:         SELECT CASE SelectedJob
718:             CASE 5    '新規ファイル作成
719:                 CALL NewFile(p%)
720:             CASE 8    '終了
721:                 EXIT DO
722:             CASE ELSE
723:                 CLS 0
724:                 PRINT "指定した辞書ファイルがまだつくられていないか,    "
725:                 PRINT "ファイルの初期化がすんでいません. "
726:                 PRINT "「新規ファイル作成」か「終了」を選択してください. "
727:                 PRINT : PRINT
728:                 LOCATE CommandRow, CommandCol
729:                 PRINT "何かキーを押してください. "
730:                 DO
731:                 LOOP UNTIL INKEY$ <> ""
732:
733:         END SELECT
734:     END IF
735:
736: LOOP
737:
738: TIMER OFF
739: CLS
740: LOCATE CommandRow, CommandCol
741: PRINT "ただいま終了処理をしています. しばらくおまちください. "
742:
743:
744: x(0).left = p%
745: OPEN DicName$ FOR RANDOM AS #1 LEN = LEN(xdata)
746:     FOR j = 0 TO ListSize
747:         PUT #1, j + 1, x(j)
748:     NEXT j
749: CLOSE #1
750:
751: CLS 0
752:
753: END SUB
754:
755: SUB NewFile (p%)
```

```
756:
757:     CLS 0
758:
759:     PRINT "新規辞書ファイルを作成します."
760:     PRINT "古い辞書ファイルの内容は消去されます."
761:     PRINT "よろしいですか?(y/n)"
762:     DO
763:         a$ = UCASE$(INKEY$)
764:     LOOP UNTIL (a$ = "Y" OR a$ = "N")
765:     IF a$ = "N" THEN EXIT SUB
766:
767:     CALL InitArray
768:     CALL InitFreeList
769:
770:     p% = 0
771:
772:     CALL Entry(p%)
773:
774: END SUB
775:
776: SUB PrtFile (p%)
777:
778:     CLS 0
779:
780:     PRINT "辞書ファイルを印刷します."
781:     PRINT "プリンタの準備をしてください."
782:     PRINT "よろしいですか?(y/n)"
783:     DO
784:         a$ = UCASE$(INKEY$)
785:     LOOP UNTIL (a$ = "Y" OR a$ = "N")
786:     IF a$ = "N" THEN EXIT SUB
787:
788:     PRINT "印刷したい単語の範囲を指定します."
789:
790:     DO
791:         INPUT "範囲の最初の頭文字を入力してください."; u$
792:         INPUT "      最後の頭文字を入力してください."; l$
793:         u$ = UCASE$(LEFT$(u$, 1))
794:         l$ = UCASE$(LEFT$(l$, 1))
795:         IF u$ > l$ THEN
796:             PRINT "範囲指定がおかしいです."
797:         ELSE
798:             EXIT DO
799:         END IF
800:     LOOP
801:
802:     LPRINT SPC(10); : LPRINT "＊＊＊  登録単語リスト  ＊＊＊"
803:     LPRINT
804:
805:     CALL PrintTree(p%, 0, u$, l$)
806:
807: END SUB
808:
809: SUB Translate (p%)
810:
811: DIM OneWord AS Node
812: DIM Spell(1 TO InfSize) AS STRING * 1
813:
814: DO
815:     CLS
816:
817:     LOCATE JobTitleRow, JobTitleCol
818:     PRINT "＊＊＊＊＊  意味検索  ＊＊＊＊＊"
819:
820:
821:     LOCATE WordRow, WordCol - 6
822:     PRINT "単語:"
823:     LINE (100, 140)-(314, 163), , B
824:
```

```
825:      LOCATE MeanRow, MeanCol - 6
826:      PRINT "意 味 : "
827:      LOCATE MeanRow, MeanCol
828:      LINE (1, 204)-(620, 227), , B
829:
830:      LOCATE CommandRow, CommandCol
831:      PRINT "単 語 の ス ペ ル を 入 れ て く だ さ い 。"
832:      LOCATE CommandRow + 1, CommandCol
833:      PRINT "( 終 了 す る 時 は ピ リ オ ド ( . ) を 入 力 ) " + SPACE$(40)
834:
835:      s$ = ""
836:      CALL ReadLine(s$, InfSize, WordCol, WordRow)
837:      OneWord.inf = s$
838:      IF INSTR(OneWord.inf, ".") THEN EXIT SUB
839:
840:      LOCATE WordRow, WordCol
841:
842:      LengthWord = LEN(RTRIM$(OneWord.inf))
843:      IF LengthWord > InfSize THEN LengthWord = InfSize
844:
845:      FOR I = 1 TO LengthWord
846:          Spell(I) = MID$(OneWord.inf, I, 1)
847:          PRINT Spell(I);
848:      NEXT I
849:
850:      s$ = LEFT$(OneWord.inf, 20)
851:      where = Search(s$, p%)
852:      IF where <> 0 THEN
853:          AccentPos = x(where).acc
854:
855:          IF AccentPos <> 0 THEN
856:              LOCATE WordRow, WordCol + AccentPos - 1
857:              COLOR 4
858:              PRINT Spell(AccentPos)
859:              COLOR 7
860:          END IF
861:
862:          LOCATE MeanRow, MeanCol
863:          PRINT x(where).synop
864:      ELSE
865:          LOCATE MeanRow, MeanCol
866:          COLOR 5
867:          PRINT "こ の 単 語 は 登 録 さ れ て い ま せ ん 。"
868:          COLOR 7
869:      END IF
870:
871:      LOCATE CommandRow, CommandCol
872:      PRINT "何 か キ ー を 押 し て く だ さ い 。 " + SPACE$(40)
873:      LOCATE CommandRow + 1, CommandCol
874:      PRINT SPACE$(40)
875:
876:      DO
877:      LOOP UNTIL INKEY$ <> ""
878: LOOP
879:
880: END SUB
881:
```

［8/終了］を選ぶと，メモリー上の配列x( )の内容を，フロッピィディスク上のランダムファイルに書き戻す処理を行って，このプロシージャを終わっています．

2.3.5.3 FUNCTIONプロシージャ DispMenu

単語帳のメニュー表示機能をはたす関数です．単語帳の8つの機能をメニュー表示し，カーソルキーを上下させて機能を選択するようにしています．この部分の技法は**2.1**で，印刷するファイルをカーソルで選ばせるときと同じ方法を使っています．このメニューではさらに，カーソルキーを使わないでも，機能名の前に表示された1～8の数字をキー入力しても，即座にその機能が実行できる，ショートカットキー入力を受けつけるようになっています．このプロシージャは関数ですので，コールしたプロシージャに，単語帳の機能に対応した1～8の数値を返します．

2.3.5.4 SUBプロシージャ Entry

単語登録機能をはたすプロシージャです．このプロシージャでは，

単語のスペル → アクセント → 意味

の順に入力をさせるようになっています．

単語のスペルと意味の入力のときには，1行編集プログラムのモジュールを利用するようになっています．このモジュールの説明は**2.4**で行います．

アクセントの入力は，SUBプロシージャGetAccPosを使っています．このプロシージャについては，次項で詳しく説明します．

すでに登録済みの単語を再び登録すると，上書き処理が行われてしまうので，その前に確認を求めるようにします．

まず，入力された単語がすでに辞書ファイルに存在するかどうかを，サブモジュールDIC-AVL. BASのFUNCTIONプロシージャSearchを使って調べます．この関数は，辞書ファイルにその単語が存在しなかった場合に0を返してきます．

したがって，0でないリターン値が返ってきた場合は，上書きしてよいかどうかの確認を行ったあと，アクセントの入力に移ります．上書きがまずい場合は，ここの時点でこのプロシージャから抜けることができます．

意味の入力時に，ミススペルやアクセントの入力ミスに気づいた場合は，スラッ

シュ(/)を入力することにより，スペルの入力まで処理をさかのぼることができます．

3項目とも入力し終えて⏎キーを押すと，その単語のスペル，アクセントおよび意味のすべてが確定したとみなし，サブモジュールDIC-AVL. BASのSUBプロシージャInsertをコールすることにより，辞書ファイルにその単語を登録します．

このEntryプロシージャは全体が無限ループ構造になっており，単語のスペル入力時に，ピリオド(.)を入力すると，

```
IF INSTR(OneWord.inf,".") THEN EXIT SUB
```

によって，プロシージャを終わることができます．

2.3.5.5 SUBプロシージャ　GetAccPos

このプロシージャは，単語登録と訂正の処理を行う場合，画面上のカーソルキーを左右に移動させることによって，アクセントの位置を入力させるものです．図2.3.14にプロシージャのフローチャートを示します．

フローチャートにしたがって，順に説明していきます．

単語のスペルは引数で渡されてきますから，まずこれを文字列配列Spell( )に格納します．これは，アクセント位置を表示するのに，その位置の英文字を反転表示させるため，単語のスペルを1文字単位で配列にしまっておいて，アクセスするためです．

アクセント位置を格納しておく変数AccPosGetの初期値を，単語の先頭英文字，すなわち1にセットしておきます．そして，その英文字を反転表示します．

ここからDO/LOOP UNTILループになっています．このループは⏎キーか，ESCキーが押されると終了します．⏎キーが押された場合は，画面で反転表示されている英文字の位置がAccPosGetにセットされて返されることになります．

ESCキーが押された場合は，アクセント入力はパスされたものとみなし，AccPosGetには0がセットされ，反転表示をキャンセルしてから返されます．

ループの中は，押されたキーが←(左矢印)キーと→(右矢印)キーの場合の処理が示されています．それぞれ英単語の長さを越えない間は，矢印キーに対応して反転位置を移動させ，AccPosGetの内容を書き換えます．

●図2.3.14 SUB GetAccPosのフローチャート

文字列配列Spell()に単語のスペルをセットしておく

AccPosGet←1

AccPosGet の位置のスペルを反転表示する

a$←INKEY$

←キーまたは→キーか

False

True

←キー

→キー

NewAccentPos←AccPosGet−1

NewAccentPos←AccPosGet+1

カーソルは単語のスペル内か

False

True

New Accent Posの位置に反転表示を移動

AccPosGet←NewAccentPos

a$←INKEY$

ESCキーか

False

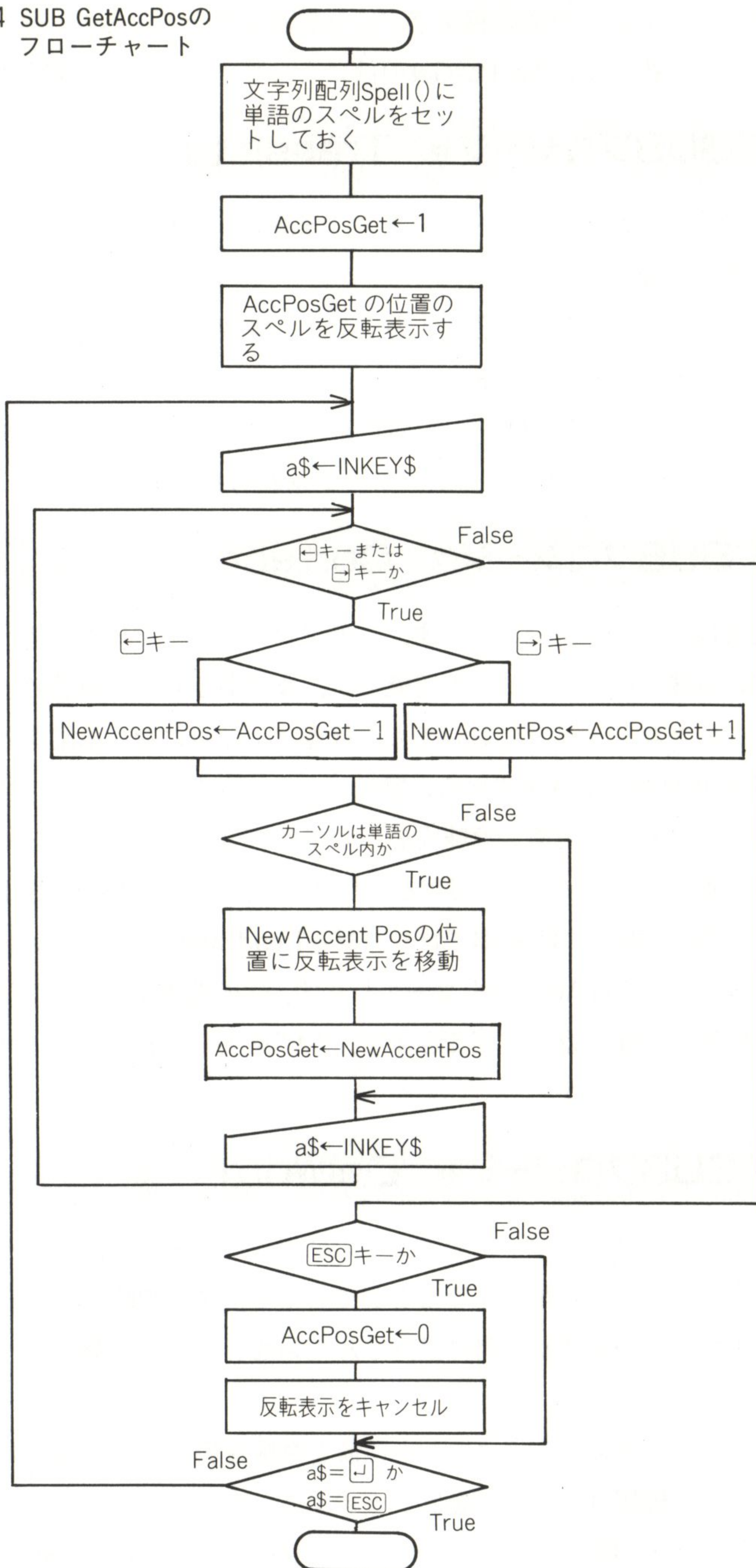
True

AccPosGet←0

反転表示をキャンセル

a$=↵ か a$=ESC

False

True

指定したいアクセント位置の英文字が反転表示されたところで，⏎キーを押すと，このループを抜けて，AccPosGetの値によってアクセント位置を返します．

2.3.5.6 SUBプロシージャ　Translate

意味検索機能をはたすプロシージャです．単語のスペルを入力すると，そのアクセント位置と訳語を表示します．

もし，辞書ファイルにない単語を入力すると，関数Searchが0を返してきますから，警告メッセージを表示します．

このプロシージャも無限DOループになっていますから，単語のスペルの代わりにピリオド(.)を入力すると，このプロシージャから抜けます．

2.3.5.7 SUBプロシージャ　Correct

訂正機能をはたすプロシージャです．このプロシージャで単語のスペル，アクセント位置，意味の訂正がどれでも可能です．まず訂正したい単語のスペルを入力します．このプロシージャにおいても，辞書ファイルにない単語を入力すると関数Searchが0を返してきますから，警告メッセージを表示します．登録済みの単語であればアクセント位置と意味を表示してサブメニューを表示します．

訂正項目の番号によって，訂正処理を行います．訂正の場合もSUB Entryと同様，スペルと意味の場合にはSUBプロシージャReadLineを，アクセントの場合にはSUBプロシージャGetAccPosをコールします．訂正処理は[4：訂正終了]を選択するまで何度でも繰り返し訂正できます．“4”を入力するとこのプロシージャから抜けることができます．

2.3.5.8 SUBプロシージャ　DelWord

削除機能をはたすプロシージャです．このプロシージャによって，不要になった単語を辞書ファイルから削除することができます．まず削除したい単語のスペルを入力します．その単語が辞書に登録されていないときには警告メッセージを表示します．

登録されている場合には，アクセント位置と意味を表示すると同時に，削除してよいかどうかの確認メッセージを出します．

もし，削除してよい場合には，サブモジュールDIC-AVL. BASのSUBプロシー

ジャDeleteをコールして，辞書ファイルから削除します．

このプロシージャを抜けるには，単語のスペルを入力する代わりにピリオド(.)を入力します．

2.3.5.9 SUBプロシージャ　NewFile

辞書ファイルを初期化する場合にこのプロシージャをコールします．プロシージャの先頭で誤ってこのプロシージャをコールした場合にそなえ，警告メッセージを表示します．

辞書ファイルの初期化は，AVL-木の初期化とp%を0に設定しなおすことを行います．AVL-木関係の初期化は，SUBプロシージャInitArrayおよびサブモジュールDIC-AVL. BASのSUBプロシージャInitFreeListをコールすることによって行います．

2.3.5.10 SUBプロシージャ　DispFileとPrtFile

この2つのプロシージャは，辞書ファイルの内容を指定した範囲についてアルファベット順に，プロシージャDispFileはディスプレイ画面に，プロシージャPrtFileはプリンタに印刷させるものです．

どちらのプロシージャもまず表示する範囲を入力させて，それらを引数にして，プロシージャDispFileではサブモジュールDIC-AVL.BASのSUBプロシージャDispTreeを，プロシージャPrtFileでは同じサブモジュールのSUBプロシージャPrintTreeをコールします．具体的な処理はコールしたプロシージャのほうで行っています．

2.3.6 エラートラッピングとイベントトラッピング

本プログラムでは，第1部の**1.4.4**でも紹介したエラートラッピングとイベントトラッピングを利用しています．以下にその概要を説明します．

2.3.6.1 エラートラッピング

QBプログラムでエラートラッピングを行うためには，ON ERROR GOTOステートメントとエラー処理ルーチンが必要です．

エラー処理ルーチンは，モジュールレベルコードのENDステートメントのあとの，行ラベルERREXIT以後に書かれています．

このプログラムでは，辞書ファイルはBドライブにセットするようになっていますが，フロッピィディスクの入れ間違いなどで，そのドライブに辞書ファイルが存在しない場合，この処理ルーチンが働いてエラーメッセージを表示し，正しくフロッピィディスクを差し込むように促します．正しくセットされると，エラーが発生した文に実行が戻るようになっています．

そのほかのエラーの場合は，エラー番号を表示したあと，ファイルをクローズして，プログラムの実行を終了するようになっています．

2.3.6.2 イベントトラッピング

このプログラムでは，イベントトラッピングのうち，タイマートラッピング処理を行っています．

パーソナル辞書の使用中のディスプレイ画面に，システム関数を利用して，年月日と1秒ごとの時刻表示を行うものです．

メインモジュールのモジュールレベルコードに，次のようなステートメントを書いて置いておきます．

```
ON TIMER(1) GOSUB DispTime
```

また，モジュールレベルコードには，イベント処理サブルーチンを記述しておきます．

そして，このイベントを実行したい場所で，TIMER ONコマンドを記述すれば，INPUT文やPRINT文でプログラムの実行が止まったり，TIMER OFFコマンドでイベントを中止しないかぎり，画面の指定した位置に時刻を表示し続けます．

2.3.7 プロシージャのクロスリファレンス

メインモジュールの各プロシージャについての説明は，前節で行いました．

しかし，本プログラムは3個のモジュールから成り立っているため，メインモジュールから，サブモジュールの多くのプロシージャをコールしています．

パーソナル単語帳におけるサブモジュールの説明は，**2.4**以降で説明しますが，複数のモジュールからなるプログラムの場合，プロシージャ間の関係が複雑になりますので，**表2.3.1**にメインモジュールに関係するプロシージャのクロスリファレンスを示します．表の縦軸には呼び出し側のプロシージャ名と所属しているモジュール名が，横軸には呼び出される側のプロシージャ名と所属モジュール名が記入されており，プロシージャコールが行われている個所には○印が記入されています．○*印は，プロシージャが，自分自身を再帰的に呼び出していることを示します．

多数のプロシージャよりなるプログラムを作成した場合，このようなクロスリファレンスは，あとでデバッグを行う場合にたいへん役にたつものです．

クロスリファレンスの作成は，紙の上で行うと，罫線を引くだけでも面倒な作業となるので，表計算ソフトウェア，たとえばロータス1-2-3やマルチプランなどで行うとよいでしょう．

●表2.3.1　プロシージャのクロスリファレンス

呼び出し側 ＼ 呼び出される側		メインモジュール				
モジュール		Correct	DelWord	DispFile	DispMenu	
DIC-MAIN	モジュールレベルコード					
↑	Correct					
↑	DelWord					
↑	DispFile					
↑	DispMenu					
↑	Entry					
↑	GetAccPos					
↑	InitArray					
↑	Menu	○	○	○	○	
↑	NewFile					
↑	PrtFile					
↑	Translate					
DIC-EDIT	DeleteStr					
↑	InsertStr					
↑	IsKanji					
↑	ReadLine					
DIC-AVL	Delete					
↑	DispTree					
↑	InitFreeList					
↑	Insert					
↑	PrintTree					
↑	Search					

呼び出し側 ＼ 呼び出される側		サブモジュール				
モジュール		DeleteStr	InsertStr	IsKanji	ReadLine	
DIC-MAIN	モジュールレベルコード					
↑	Correct				○	
↑	DelWord				○	
↑	DispFile					
↑	DispMenu					
↑	Entry				○	
↑	GetAccPos					
↑	InitArray					
↑	Menu					
↑	NewFile					
↑	PrtFile					
↑	Translate					
DIC-EDIT	DeleteStr					
↑	InsertStr					
↑	IsKanji					
↑	ReadLine	○	○	○		
DIC-AVL	Delete					
↑	DispTree					
↑	InitFreeList					
↑	Insert					
↑	PrintTree					
↑	Search					

○* 再帰的呼び出し

Entry	GetAccPos	InitArray	Menu	NewFile	PrtFile	Translate
			○			
	○					
	○					
○				○	○	○
○		○				

サブモジュール

Delete	DispTree	InitFreeList	Insert	PrintTree	Search
			○		○
○					○
	○				
			○		○
		○			
				○	
					○
	○*				
				○*	

2.4 1行編集プログラム

2.4.1 プログラムの概要

単語のスペルおよび意味の入力や訂正など，1行の入力を行いたい場合，通常は1行入力コマンドLINE INPUTを用います．しかし，LINE INPUTでは次のような不都合があります．

文字を入力中に，誤って[↑](上矢印)キーや[↓](下矢印)キーを押してしまうと，入力位置がずれてしまい，文字列が正しく入力できなくなるおそれがあります．

また，すでに入力済みの文字列を画面に表示させて，一部訂正したい場合も，この方法ではできません．

それを解消するためには，面倒でも自分でプログラムするしかありません．

ここでは，1行ではありますが，市販のエディタのような機能をそなえた1行編集プログラムを作成してみました．つまり，左右矢印キーによるカーソルの移動，[DEL]キーによるカーソル位置の1文字消去，[BS]キーによるカーソル位置の左文字の削除，およびカーソル位置への文字列の挿入といった機能をそなえた1行エディタです．

このような1行エディタは，ほかのプログラムにおいても有用なユーティリティになると考え，パーソナル単語帳とは切り放し，独立したモジュールとして作成しました．したがって，プログラム作成時に，1行編集の機能が必要になった場合，このモジュールを連結して必要な引数を受け渡しすれば，簡単に目的が達成できます．

通常，このようなプロシージャは，画面のアドレスを直接アクセスするために，機械語でプログラムされます．しかし，ここでは，あえてQBで書いてみました．結果的には満足するものが得られました．

モジュールは3つのSUBプロシージャと1つのFUNCTIONプロシージャより

なります．1行編集プログラムの本体となるプロシージャはSUB ReadLineで，ほかのモジュールからこのプログラムを呼ぶためには，そのモジュールのモジュールレベルコードにおいて，次に示すDECLARE文が必要です．

```
DECLARE SUB ReadLine (s1$, BufLen, x0, y0)
```

また，実際に1行エディタが必要な場所では，このSUBプロシージャReadLineをCALLします．

4つの引数は次のようです．すなわち，BufLenはエディタで編集を行う文字数(半角)，x0とy0はエディタ行の左端の画面上のカラムと行位置，そしてs1$が編集する文字列です．

たとえば，画面の10行目5カラムから40文字をエディタにして，文字列s1$を編集したい場合は，

```
CALL ReadLine (s1$, 40, 5, 10)
```

といったように，1行エディタをCALLすればよいのです．では，プロシージャReadLineの内容を具体的に見ていきましょう．

2.4.2 SUBプロシージャ　ReadLine

このようなエディタプログラムの要点は次のようなことです．すなわち，カーソルキー(矢印キー)に対応して移動すること，編集対象の範囲外(ここでは，1行の範囲)にはカーソルが飛び出さないこと，漢字と半角文字とが混ざってもカーソルは正しく移動して編集可能なこと，特定のコード(ここでは⏎キー(＝13)とESCキー(＝27))を除いては，すべてのコードが入力できることなどです．具体的には次のように実現されます．

モジュールDIC-EDIT. BASは，3個のSUBプロシージャ，ReadLine, DeleteStrおよびInsertStrと，1個のFUNCTIONプロシージャIsKanjiよりなります．全体のリストをリスト2.4.1に示します．

SUBプロシージャReadLineは，1行編集プログラムの本体ともいうべきもので，図2.4.1にフローチャートを示します．

●リスト 2.4.1 サブモジュールDIC-EDIT・BASの全リスト

```
 1: DECLARE SUB DeleteStr (s1$, start%, Num%)
 2: DECLARE SUB InsertStr (s2$, s1$, PosNum%)
 3: DECLARE FUNCTION IsKanji! (c$)
 4: '
 5: CONST False = 0, True = NOT False
 6:
 7: SUB DeleteStr (s1$, start%, Num%)
 8:
 9:     IF (start% - 1) >= 0 AND (LEN(s1$) - start% - Num% + 1) >= 0 THEN
10:
11:         s1$ = LEFT$(s1$, start% - 1) + RIGHT$(s1$, LEN(s1$) - start% - Num% + 1)
12:
13:     END IF
14:
15:
16: END SUB
17:
18: SUB InsertStr (s2$, s1$, PosNum%)
19:
20:     s1$ = LEFT$(s1$, PosNum% - 1) + s2$ + MID$(s1$, PosNum%)
21:
22: END SUB
23:
24: FUNCTION IsKanji (c$)
25:   IF (c$ >= CHR$(&H80) AND c$ <= CHR$(&H9F)) OR (c$ >= CHR$(&HE0) AND c$ <= CHR$(&HFF))
    THEN
26:     IsKanji = True
27:   ELSE
28:     IsKanji = False
29:   END IF
30: END FUNCTION
31:
32: SUB ReadLine (s1$, BufLen, x0, y0)
33:
34: DIM c0 AS STRING * 1, c1 AS STRING * 1, c2 AS STRING * 1, c AS STRING * 1
35: DIM i AS INTEGER
36: DIM Xpos AS INTEGER, xtemp AS INTEGER
37: CONST CR = 13, ESC = 27, BS = 8, NUL = 0
38:
39:
40:     c1 = " ": i = 1: Xpos = x0
41:
42:     LOCATE y0, x0
43:     PRINT s1$
44:
45:     DO WHILE c1 <> CHR$(CR)
46:
47:           IF LEN(s1$) >= BufLen THEN
48:             DO
49:             LOOP UNTIL INKEY$ = CHR$(13)
50:             IF IsKanji(MID$(s1$, BufLen, 1)) THEN
51:                 s1$ = LEFT$(s1$, BufLen - 1)
52:             END IF
53:             EXIT DO
54:           END IF
55:
56: Again:
57:         LOCATE y0, Xpos, 1
58:
59:         DO
60:             a$ = INKEY$
61:         LOOP UNTIL a$ <> ""
62:
63:         IF LEN(a$) = 2 THEN
64:             c1 = LEFT$(a$, 1)
65:             c2 = RIGHT$(a$, 1)
66:         ELSE
```

```
 67:            c1 = a$
 68:            c2 = ""
 69:        END IF
 70:
 71:
 72:        IF (c1 = CHR$(NUL) AND c2 = "K") THEN ' <- key
 73:            xtemp = Xpos - 2
 74:            IF xtemp < x0 THEN xtemp = x0
 75:            c = MID$(s1$, xtemp - x0 + 1, 1)
 76:            IF IsKanji(c) THEN Xpos = Xpos - 2 ELSE Xpos = Xpos - 1
 77:            IF Xpos < x0 THEN Xpos = x0
 78:            LOCATE y0, Xpos, 1
 79:            GOTO Again
 80:        END IF
 81:
 82:        IF c1 = CHR$(NUL) AND c2 = "M" THEN ' ->  key
 83:            c = MID$(s1$, Xpos - x0 + 1, 1)
 84:            IF IsKanji(c) THEN Xpos = Xpos + 2 ELSE Xpos = Xpos + 1
 85:            IF (Xpos - x0) > LEN(s1$) THEN
 86:                IF IsKanji(c) THEN Xpos = Xpos - 2 ELSE Xpos = Xpos - 1
 87:                LOCATE y0, Xpos, 1
 88:                GOTO Again
 89:            END IF
 90:        END IF
 91:
 92:        IF (c1 = CHR$(NUL) AND c2 = "S") THEN  ' DEL key
 93:
 94:            c = MID$(s1$, Xpos - x0 + 1, 1)
 95:            IF IsKanji(c) THEN
 96:                CALL DeleteStr(s1$, Xpos + 1 - x0, 2)
 97:            ELSE
 98:                CALL DeleteStr(s1$, Xpos + 1 - x0, 1)
 99:            END IF
100:            LOCATE y0, x0, 0: PRINT SPACE$(LEN(s1$) + 2);
101:            LOCATE y0, x0, 0: PRINT s1$;
102:            LOCATE y0, Xpos, 1
103:            GOTO Again
104:        END IF
105:
106:        IF (c1 = CHR$(BS)) THEN '  BS key
107:            s1last$ = s1$
108:            xlast = Xpos
109:            xtemp = Xpos - 2
110:            IF xtemp < x0 THEN xtemp = x0
111:            c = MID$(s1$, xtemp - x0 + 1, 1)
112:            IF IsKanji(c) THEN Xpos = Xpos - 2 ELSE Xpos = Xpos - 1
113:            IF Xpos < x0 THEN Xpos = x0
114:
115:            s1$ = LEFT$(s1$, Xpos - x0) + MID$(s1$, xlast - x0 + 1)
116:
117:            LOCATE y0, x0, 0: PRINT SPACE$(LEN(s1last$));
118:            LOCATE y0, x0, 0: PRINT s1$;
119:            LOCATE y0, Xpos, 1
120:
121:            GOTO Again
122:        END IF
123:
124:
125:        IF c1 = CHR$(NUL) THEN
126:            GOTO Again
127:        ELSE
128:            s2$ = a$
129:        END IF
130:
131:        IF c1 <> CHR$(CR) THEN
132:            IF Xpos = (x0 + LEN(s1$)) THEN
133:                s1$ = s1$ + s2$
134:            ELSE
135:                CALL InsertStr(s2$, s1$, Xpos - x0 + 1)
```

```
136:              END IF
137:              LOCATE y0, x0, 0: PRINT SPACE$(LEN(s1$));
138:              LOCATE y0, x0, 0: PRINT s1$;
139:              Xpos = Xpos + LEN(s2$): LOCATE y0, Xpos, 1
140:          END IF
141:
142:      LOOP
143:
144: END SUB
145:
```

このプロシージャは，全体が大きなDO WHILE/LOOPになっており，文字列の編集が終了して⏎キーが押されるまで,編集を続けられるようになっています．

また，ループの先頭の部分では，次に示すように，エディタの編集領域，すなわちパラメータBufLenを越えて，文字列を挿入しようとした場合は，入力された文字列を無効にするようにしています．また，⏎キーが押され編集が終了したときに，図2.4.2に示すように，エディタの最大編集文字数の位置が漢字コードの1バイト目にかかってしまった場合,そのままs1$を返してしまうとその文字が正しく表示されません．そこで関数IsKanjiを使ってチェックを行い，もしそのような場合には,最後の漢字1文字をs1$から削除してから返すようにしています(図2.4.3).

```
IF LEN(s1$) >= BufLen THEN
    DO
    LOOP UNTIL INKEY$ = CHR$(13)
    IF IsKanji(MID$(s1$, BufLen, 1)) THEN
      s1$ = LEFT$(s1$, BufLen−1)
    END IF
    EXIT DO
END IF
```

ループの中身は編集用に押されたキーの種類によって，それぞれIF文で処理が分けられています．

2.1のところでも述べましたが,PC-9801では矢印キーやBSキーは2バイトのコードが割り当てられています．また関数INKEY$もそれらのキーが押された場合，2バイト文字として値を返してきますので，処理を振り分けるまえに，次のように，押されたキーが2バイトならば，2つの1バイト文字c1とc2にわけておきます．

●図2.4.1 SUB ReadLineのフローチャート

START
初期設定
C1=↵キー
True
END
False
編集文字数が設定値をこえたときの処理
↵キーが押されたか
True
False
1文字キー入力
2バイト文字なら1バイトずつに分解
←キーが押されたか
True
左へ1文字分カーソル移動
False
→キーが押されたか
True
右へ1文字分カーソル移動
False
DELキーが押されたか
True
カーソル位置の1文字削除
False
BS キーが押されたか
True
カーソル左の1文字削除
False
そのほかのファンクションキーか
Trus
False
↵キーが押されたか
True
False
カーソル位置に文字列の挿入

●図2.4.2

全角 全角 全角

	の	位	置

↑
エディタの最大文字位置

●図2.4.3

全角 全角

最後の半角はスペースとなる

	の	位	

↑
ここまでの文字が
s1$にセットされる

```
IF LEN(a$) = 2 THEN
    c1 = LEFT$(a$, 1)
    c2 = RIGHT$(a$, 1)
ELSE
    c1 = a$
    c2 = ""
END IF
```

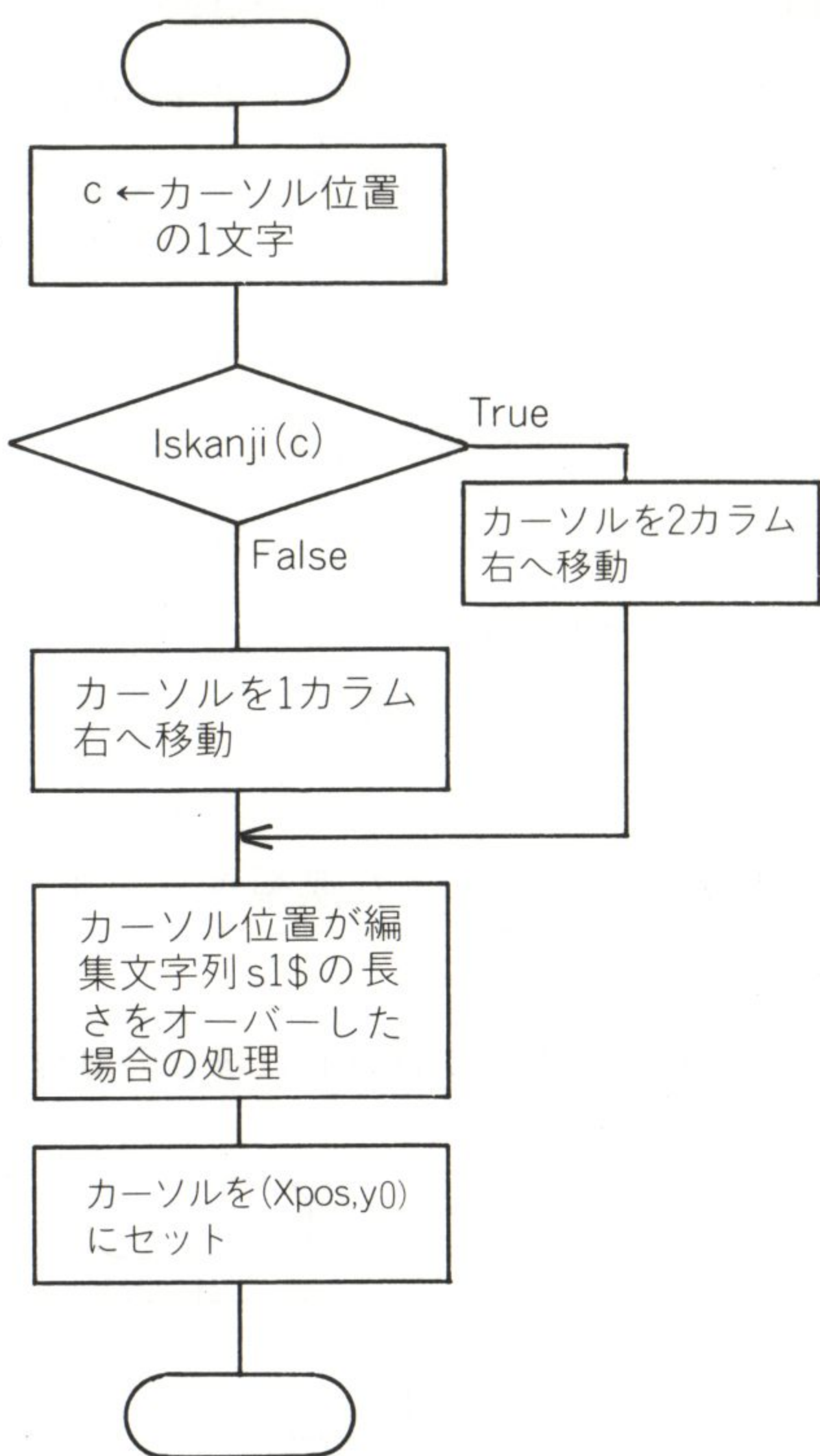
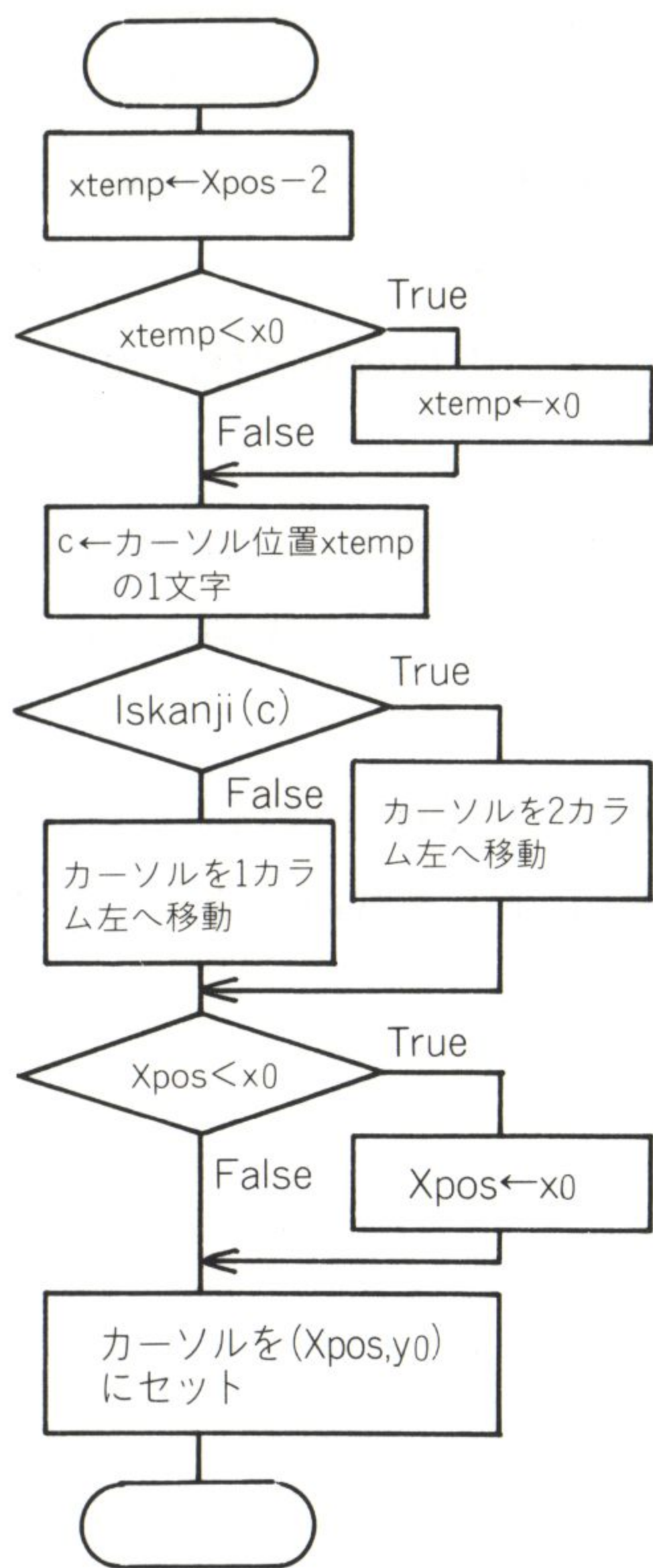
さて，エディタでキーが押される場合は大きくわけて，カーソルを移動させる場合と，文字列を削除する場合および文字列を挿入する場合が考えられます．

このエディタでは，まず矢印キーが押された場合の判断と処理を行っています．図2.4.4には，←(左矢印)キーが押された場合の処理をフローチャートで示しました．図2.4.5には，→(右矢印)キーの場合のフローチャートを示します．

どちらも共通して，編集する文字列に漢字が混じっている場合に，すこし工夫が必要です．つまり，文字列上にカーソルを移動させるとき，漢字の場合には一度に2カラム移動させる必要があるからです．このことはDELやBSによる文字列の削除の場合も同様の配慮がいります．

このプログラムでは，ある1文字が漢字コード(シフトJISコード)の1バイト目

●図2.4.4 ←キーが押された場合の処理

●図2.4.5 →キーが押された場合の処理

であるかどうかを調べる関数，FUNCTION IsKanjiを作って対応しています．

→キーの場合は，現カーソル位置の右隣の1文字を関数IsKanjiで調べます．もし，この1バイトが漢字コードの1バイト目であったとすると関数は値Trueを返してきますから，カーソルの移動量を2カラムにします．そうでないときは半角文字ですからカーソル移動量は1カラムでよいことになります．

←キーの場合は，現カーソル位置の2カラム左の1文字を関数IsKanjiで調べ，Trueならカーソル移動量を2カラム，Falseなら1カラムにします．

DELキーが押された場合の処理のフローチャートを図2.4.6に示します．このルーチンでは，SUBプロシージャDeleteStrをコールしています．ここでも，削除する文字が半角か全角かによって，削除するバイト数が異なりますので，カーソル位置の文字を関数IsKanjiで調べておき，DeleteStrをコールするときに，3番目の引

数としてバイト数を渡します.

[BS]キーが押された場合の処理のフローチャートを図2.4.7に示します.このキーの働きはカーソルの左隣の1文字を削除すると同時に,カーソルを1文字分左に移動させるというものですから,ちょうど,[←]キーの働きと[DEL]キーの働きを合わせたような処理を行うことになります.

カーソルの左隣が漢字かどうかのチェックは,[←]キーの場合と同様,現カーソルの2つ左の文字を関数IsKanjiを使って調べます.

このように,押されたキーが[←][→]キー,[DEL]キーおよび[BS]キーの場合は,IF文でチェックされて,それぞれの処理を施され,ふたたびループの先頭の行ラベルAgainに戻されて,次のキー入力を待ちます.

それ以外の2バイトコードのキー入力(1バイト目がCHR$(0)のもの)は,次に示すIF文でチェックされ無視されます.そのほかは[↵]キーを含め,挿入文字列と判断されます.

●図2.4.6 [DEL]キーが押された場合の処理

●図2.4.7　BSキーが押された場合の処理

s1last$←s1$
xlast←Xpos

xtemp←Xpos－2

xtemp＜x0
True
xtemp←x0
False

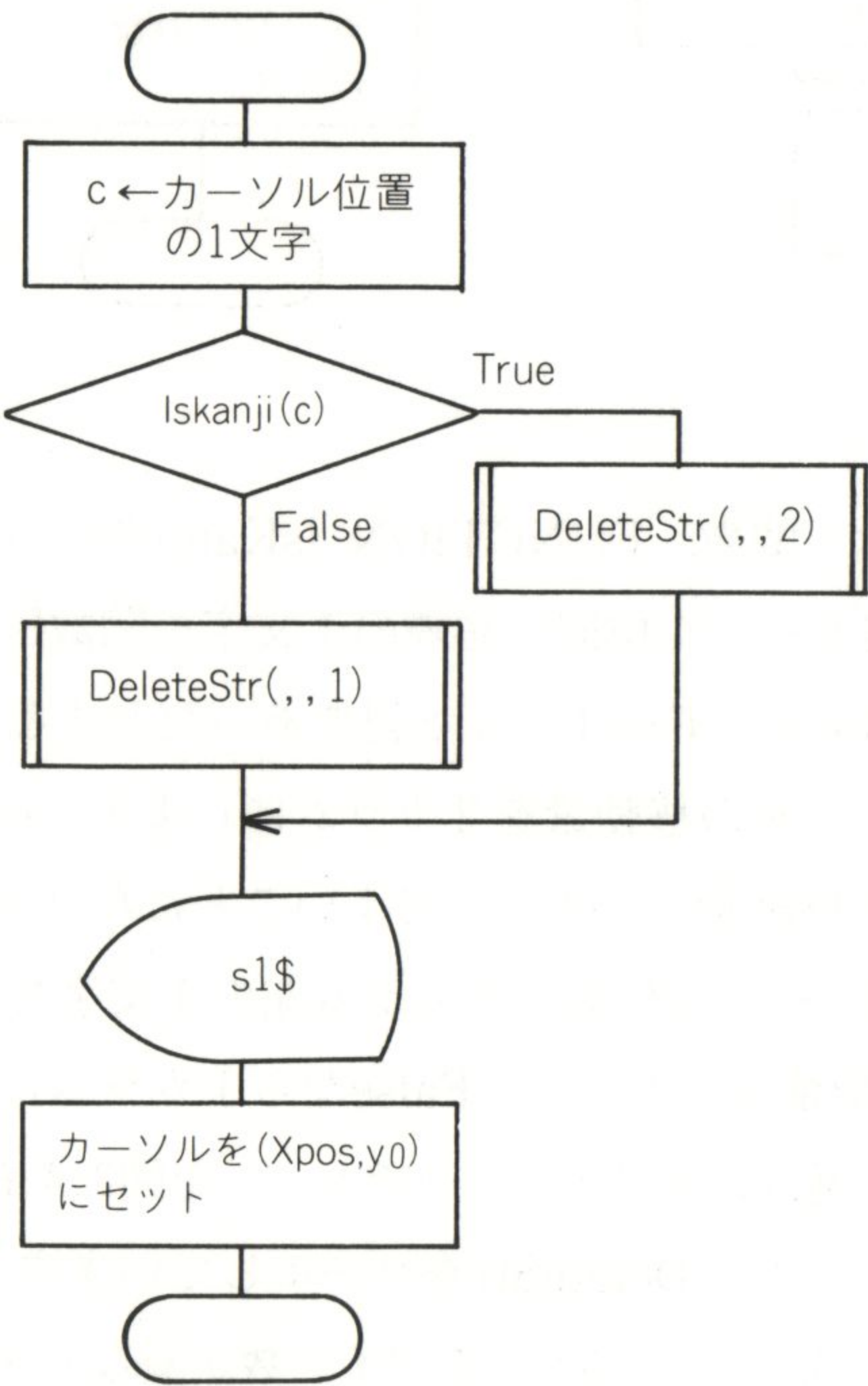
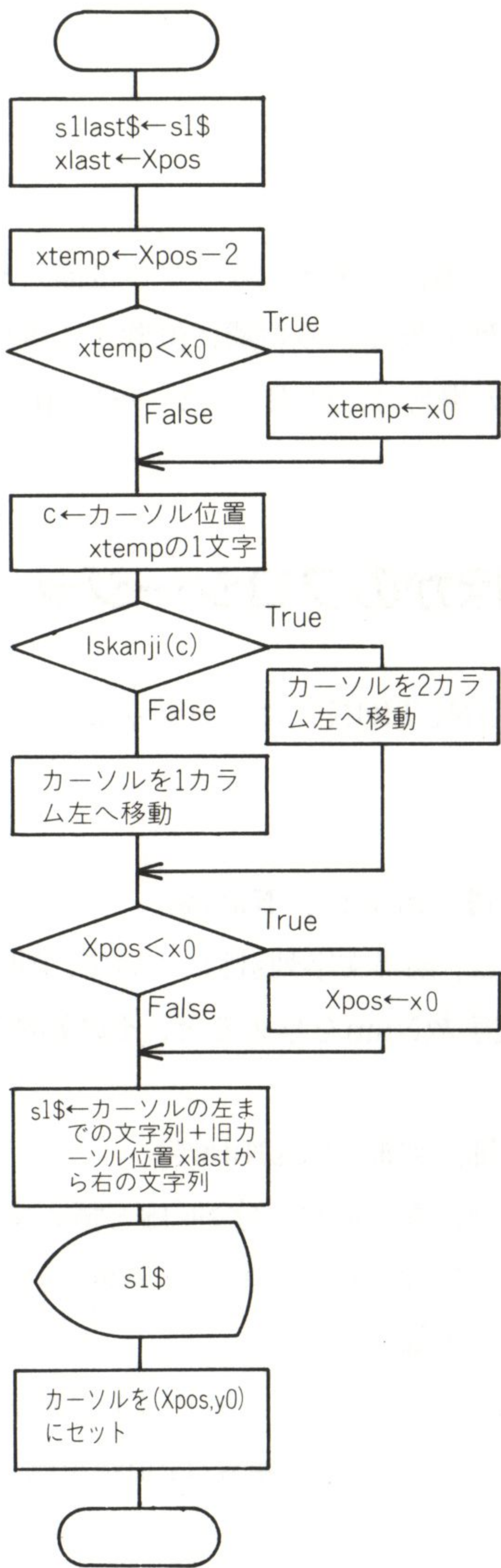
c←カーソル位置xtempの1文字

Iskanji(c)
True
カーソルを2カラム左へ移動
False
カーソルを1カラム左へ移動

Xpos＜x0
True
Xpos←x0
False

s1$←カーソルの左までの文字列＋旧カーソル位置xlastから右の文字列

s1$

カーソルを(Xpos,y0)にセット

```
IF c1 = CHR$(NUL)  THEN
   GOTO Again
ELSE
   s2$ = a$
END IF
```

押されたキーが⏎キーの場合を除き，カーソル位置にその文字を挿入します．カーソルが編集中の文字列の最後尾の右隣に位置する場合は，文字を連結するだけですが，文字列の途中の場合は，SUBプロシージャInsertStrをコールします．

2.4.3 そのほかのプロシージャ

そのほかのSUBおよびFUNCTIONプロシージャについて簡単に説明しておきます．

SUB DeleteStr(s1$, start%, Num%)

このプロシージャは，編集文字列s1$について，削除する文字列の先頭位置startと削除する文字数Numを与えると，その処理を行ったあと，結果をs1$にセットします．

SUB InsertStr(s2$, s1$, PosNum%)

このプロシージャは，挿入前の文字列s1$と挿入文字列s2$，および挿入位置を与えると，挿入処理を行ったあと，結果をs1$にセットします．

FUNCTION IsKanji(c$)

この関数は，パラメータc$が漢字コードの1バイト目かどうか判定します．その場合はTrueを，そうでない場合はFalseを返します．

2.5 ダイナミックバランス木

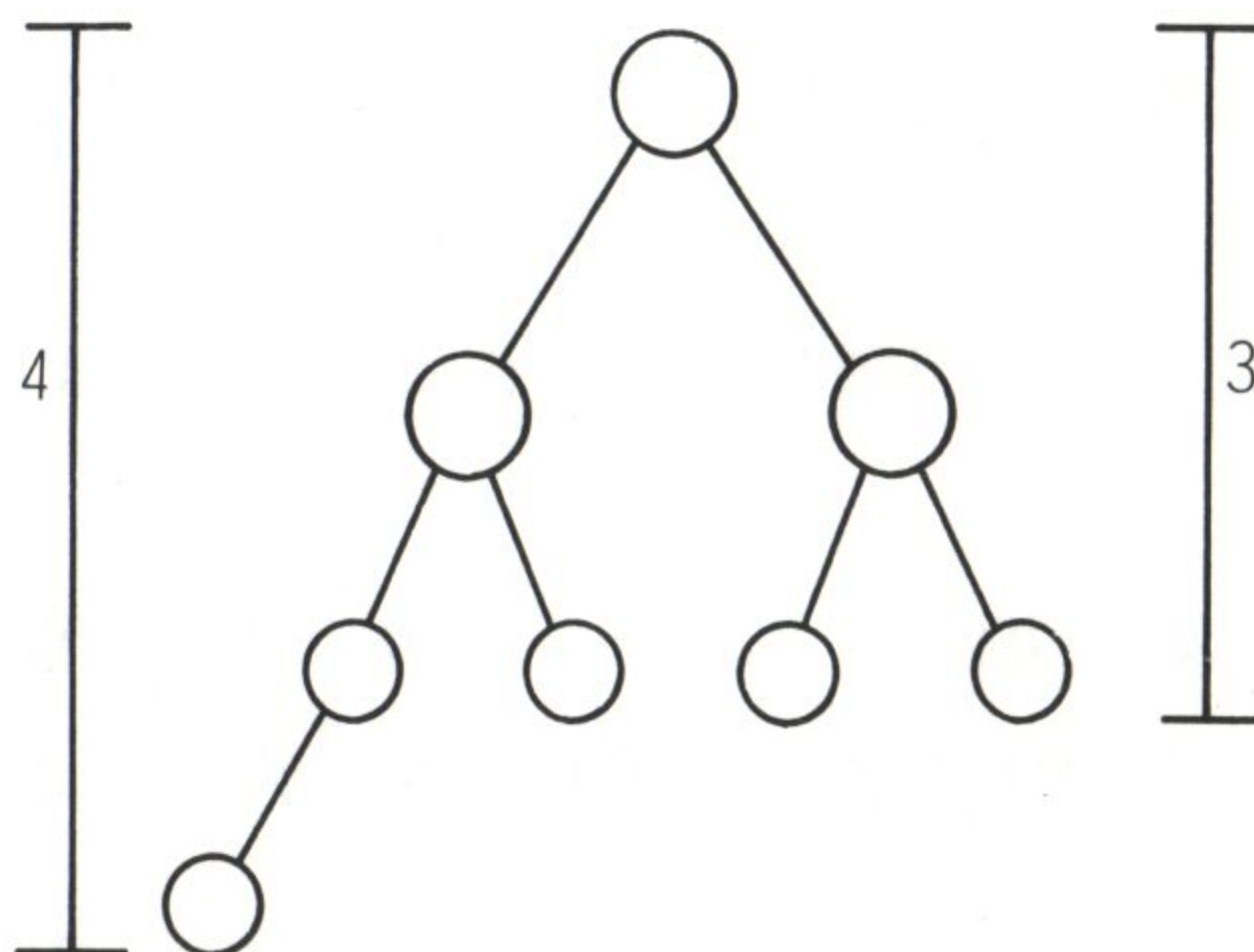
2分木による検索については**1.7**で説明しました．その際，情報が挿入される順序によっては，左右のバランスの悪い2分木ができることを例示しました．

本章では，左右のバランスがとれるように挿入や削除を実行する，ダイナミックバランス木と呼ばれる方法を説明します．これは**2.3**の単語帳で用いられているものです．

ダイナミックバランス木は考案者(Adel'son-Vel'skiiとLandis)の頭文字をとってAVL-木とも呼ばれます．本文中では以下，両方の表現を適宜用います．

2.5.1 バランス木

2分木は通常の植物の木とは逆に，一番上が根で一番下が葉です．根から葉まで，階層を下りるように順に1段ずつ下りて行きます．階層の深さを表すのに，木の高さ，あるいは単に高さを定義します．すなわち，木の高さとは根から葉までの階層の数のことです．ただし，左右の部分木のうちの最大のものを指すことにします．**図2.5.1**は左の部分木をたどると高さ4,右の部分木をたどると高さ3なので，木の高さは4となります．

●図2.5.1 高さ4の2分木

左右の部分木の高さは，等しい場合もあり等しくない場合もあります．第１部の図1.7.9は，左部分木の高さが０で，右部分木の高さが５となる極端なアンバランスな木の例です．

左右の部分木の高さの差が１以内の２分木を，バランス木と呼ぶことにします．ダイナミックバランス木とは，情報の挿入や削除によって，２分木の左右の高さの差が１以内になっていること，すなわちバランス木の状態を保った木のことを言います．

2.5.2 挿　入

バランスしている木に新しい情報を挿入します．挿入自体は，**1.7**で扱った２分木への挿入なので難しく考える必要はありません．新しい節が葉として付加されるのみです．ただし，それによりバランスがくずれることがあるので，それについて調べてみます．

挿入では，新しい節が葉として付加されるので，その葉を含む部分木の高さは変わらないか，１だけ増加します．変わらなければ，２分木全体のバランスは崩れないのでそのままにします．１だけ増加した場合には，それまで高さ１だけアンバランスであったのが，バランスする場合と，逆にますますアンバランスになる場合があります．その場合には，バランスするように２分木あるいは部分木の根を入れ換える必要があります．

これらを図で表してみます．図の中で節のそばに付した数字が－１ならば，左の部分木が右の部分木より１だけ高い，あるいは左に傾いている，０ならば，左右の部分木の高さが等しい，あるいはバランスしている，１ならば右に傾いているとします．節と節を結ぶ実線は，すでにできている２分木の一部，点線は新たに追加された葉を結ぶものとします．数字が０→１のように矢印で示されると，はじめバランスしていたのが情報の付加により右へ傾いたというように読むことにします．

図2.5.2では，情報の付加によりもともと傾いていた部分木がバランスする例です．この場合には，部分木の高さは変わらないので，これで処理は終了となります．

●図2.5.2 傾いていたのがバランスする

1→0

−1→0

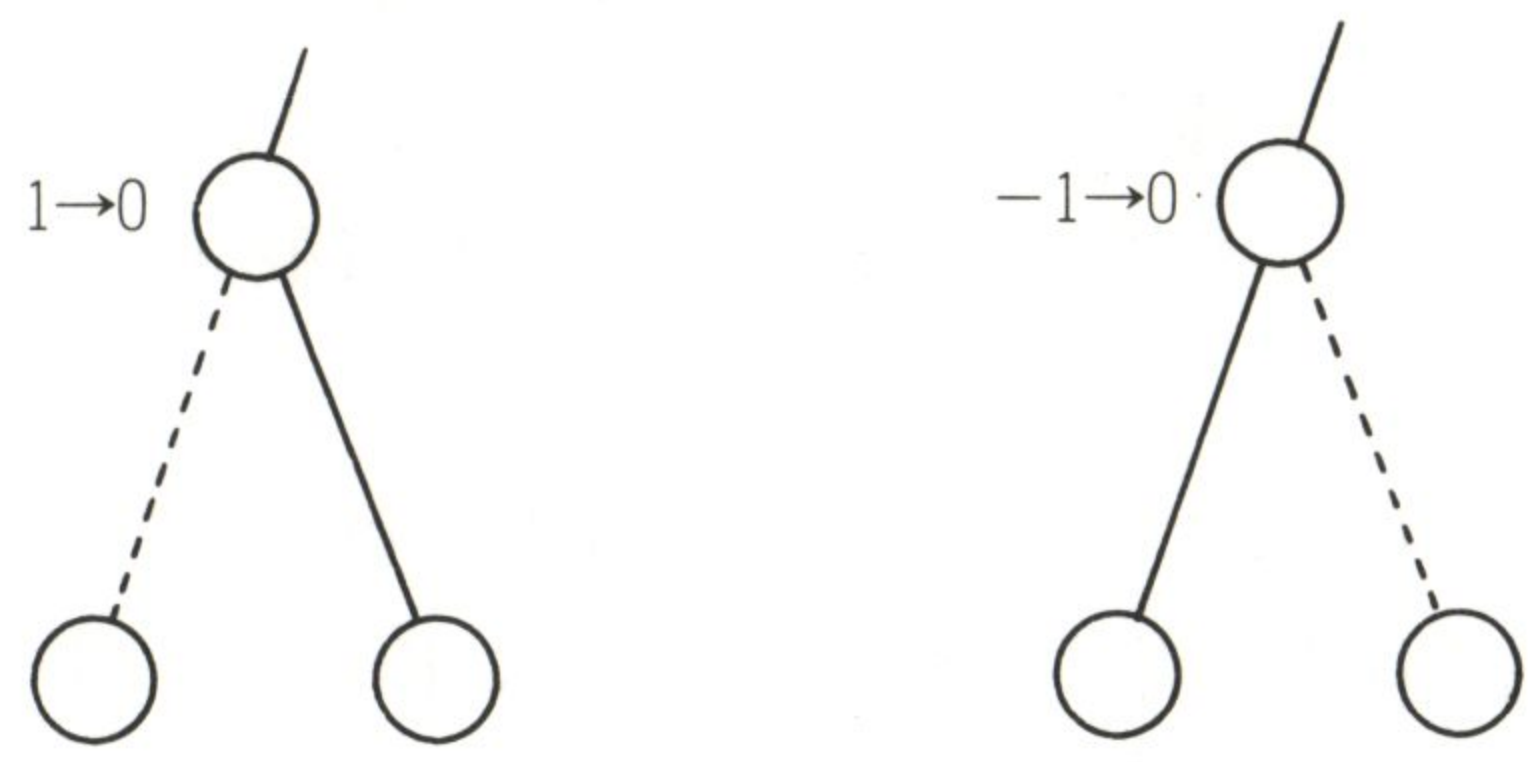
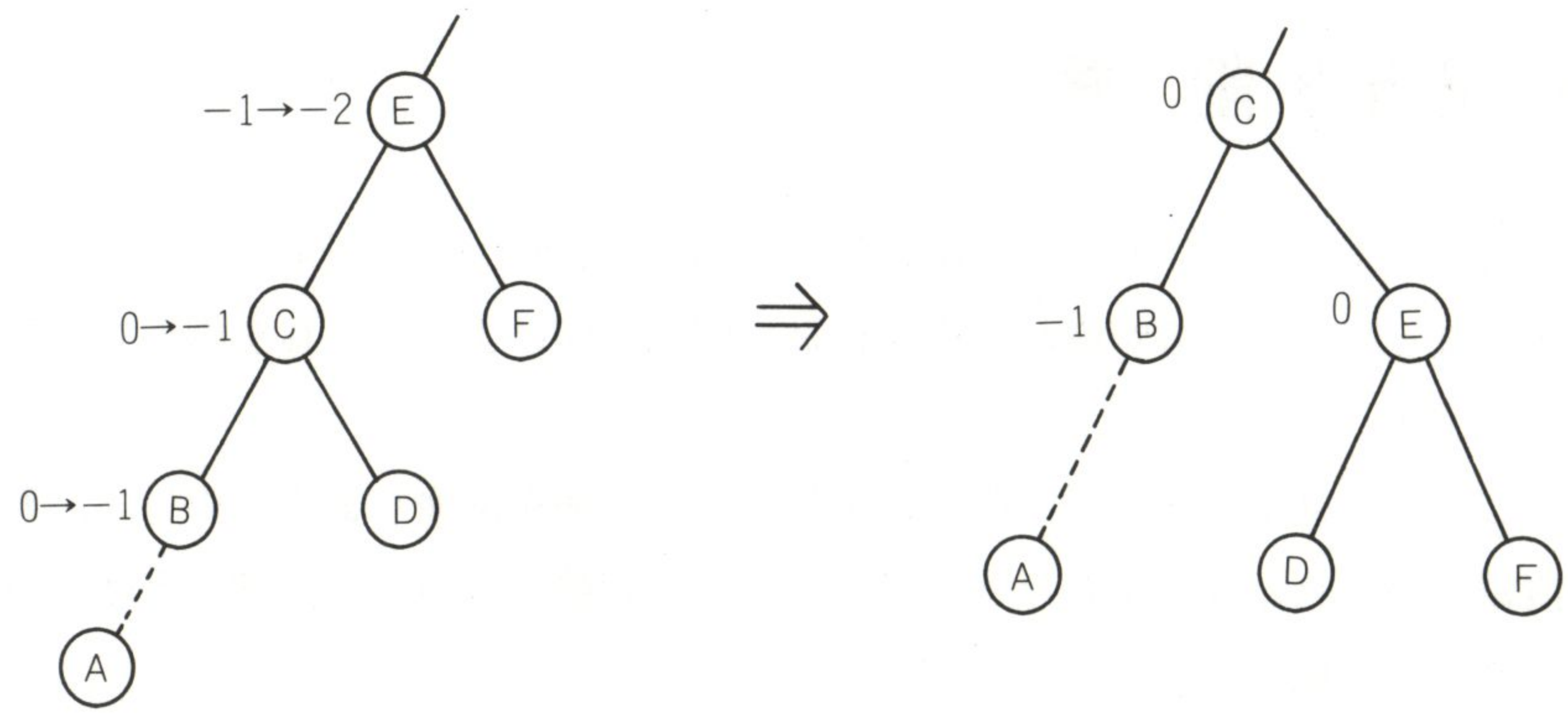



図2.5.3の例では，新しい情報Aが付加されたことにより，それまでバランスしていたBのバランスが−1となります．0から−1になるのは，高さが増加したことを意味するので，Bの親のCのバランスを調べます．Cも0から−1に変化します．やはり，高さが増加したことを意味するので，親のEのバランスを調べます．すると，Eのバランスは−1から−2となります．−2は定義してありませんが，左の部分木のほうが2だけ高いことを表すことにします．2だけ相違があることはバランス木の定義に反するので，1重回転をさせます．

この場合には，EをCの右部分木に下ろし，Cの右部分木をEの左部分木にします．そして，Eの親とCを直接結びます．図に見られるように，もとのEを根とする部分木(図では，新しくCを根とする部分木になっていますが)の高さは変わらずバランスしました．したがって，木全体のバランスすなわちバランス木としての性質は保たれます．

●図2.5.3 1重回転の例

−1→−2 E
0→−1 C
F
0→−1 B
D
A
⇒
0 C
−1 B
0 E
A
D
F

挿入によって生ずるバランスの崩れを修正するために，１重回転ではすまない場合があります．バランスが－１から－２のように崩れた節の左部分木の根の，右部分木に挿入がなされた場合には，回転によって持ち上げた節の左部分木の処理が必要となります．このタイプの回転の３つの型を図2.5.4の(a)，(b)，(c)に示します．

(a)では，Fを挿入することにより，Hのバランスが－１から－２になります．そこで，Hの左の子Cの右部分木を持ち上げると，Eの左の子Dを接続することができなくなります．そこで，Eの左の子をEの親の右の子にし，EをCの親とし，もともとCの親であったHをEの右の子にします．そして，Eの右の子はHの左の子にします．

最終的なバランスは，(a)の右の図のようにEを部分木の根とみなして，バランスは(－1，0，0)になります．（カッコの中の－１，０，０は，左の部分木のバランス，親のバランス，右の部分木のバランスを表す．）

挿入されるのが，Cの右部分木，根Fの左の部分木の場合には，(b)に示されるように回転します．(a)の場合と考え方は同じですが，Fを部分木の根とみなした場合のバランスは，(0，0，1)となります．

(c)は，このタイプの回転処理の特別な場合で，結果のバランスは(0，0，0)となります．

以上の説明では，挿入が左部分木になされたものとしましたが，右部分木の場合も同様に考えられるので説明は省略します．

2.5.3 削　除

AVL-木は２分木ですので，削除そのものは２分木のそれと同じです．ただし，削除によってバランスが崩れる場合には，回転操作によってそれを修正する必要があります．

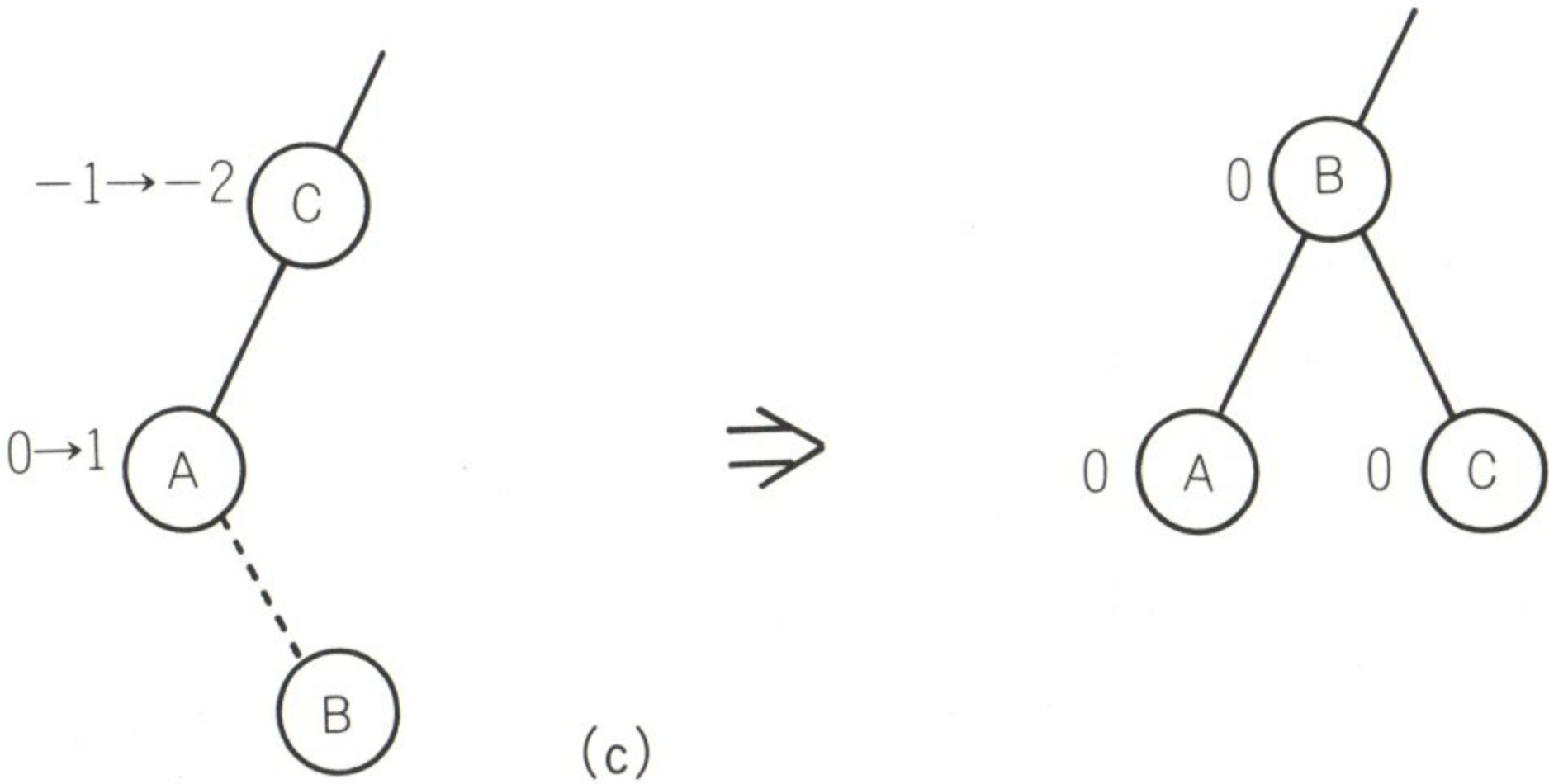
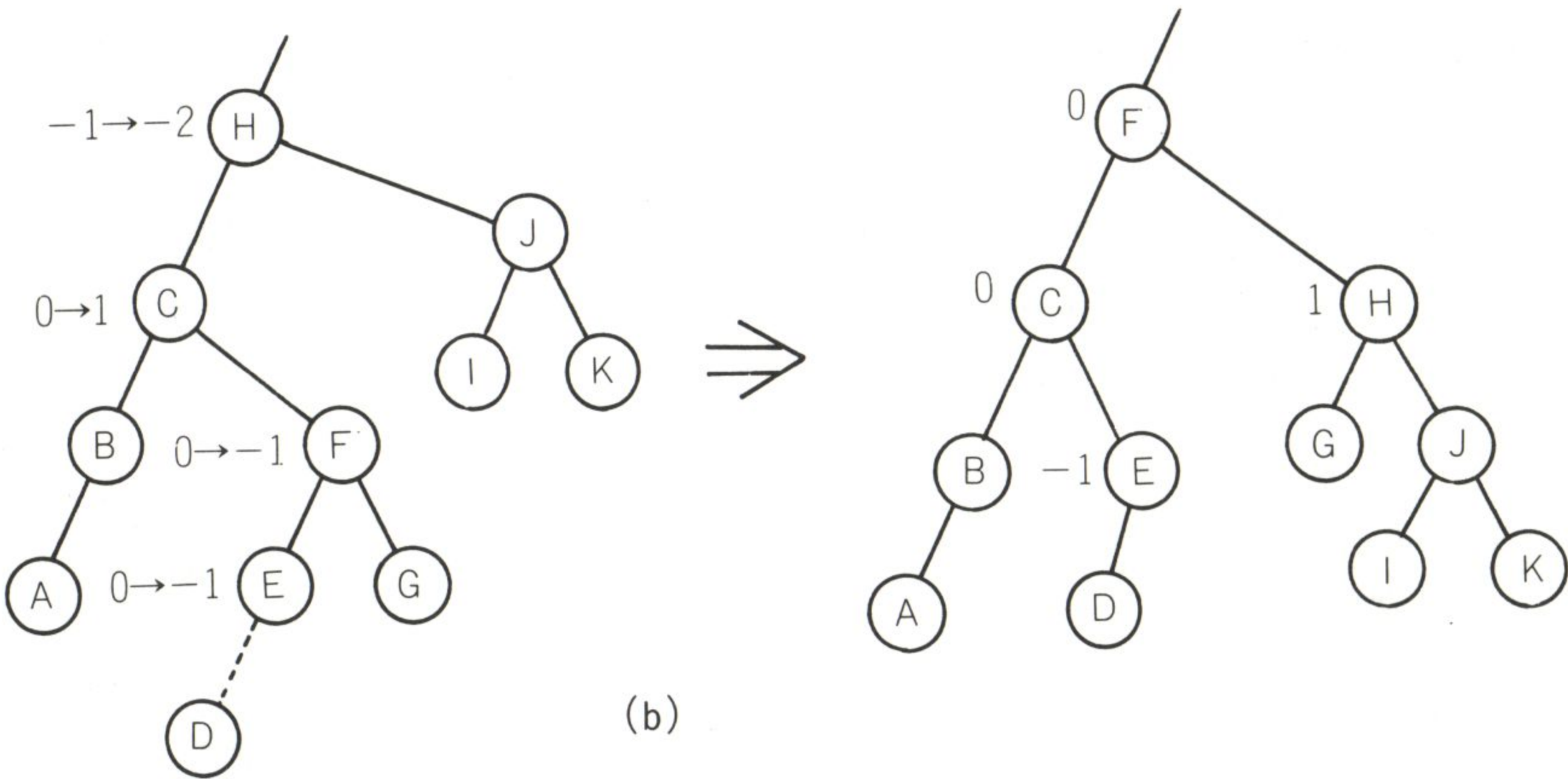
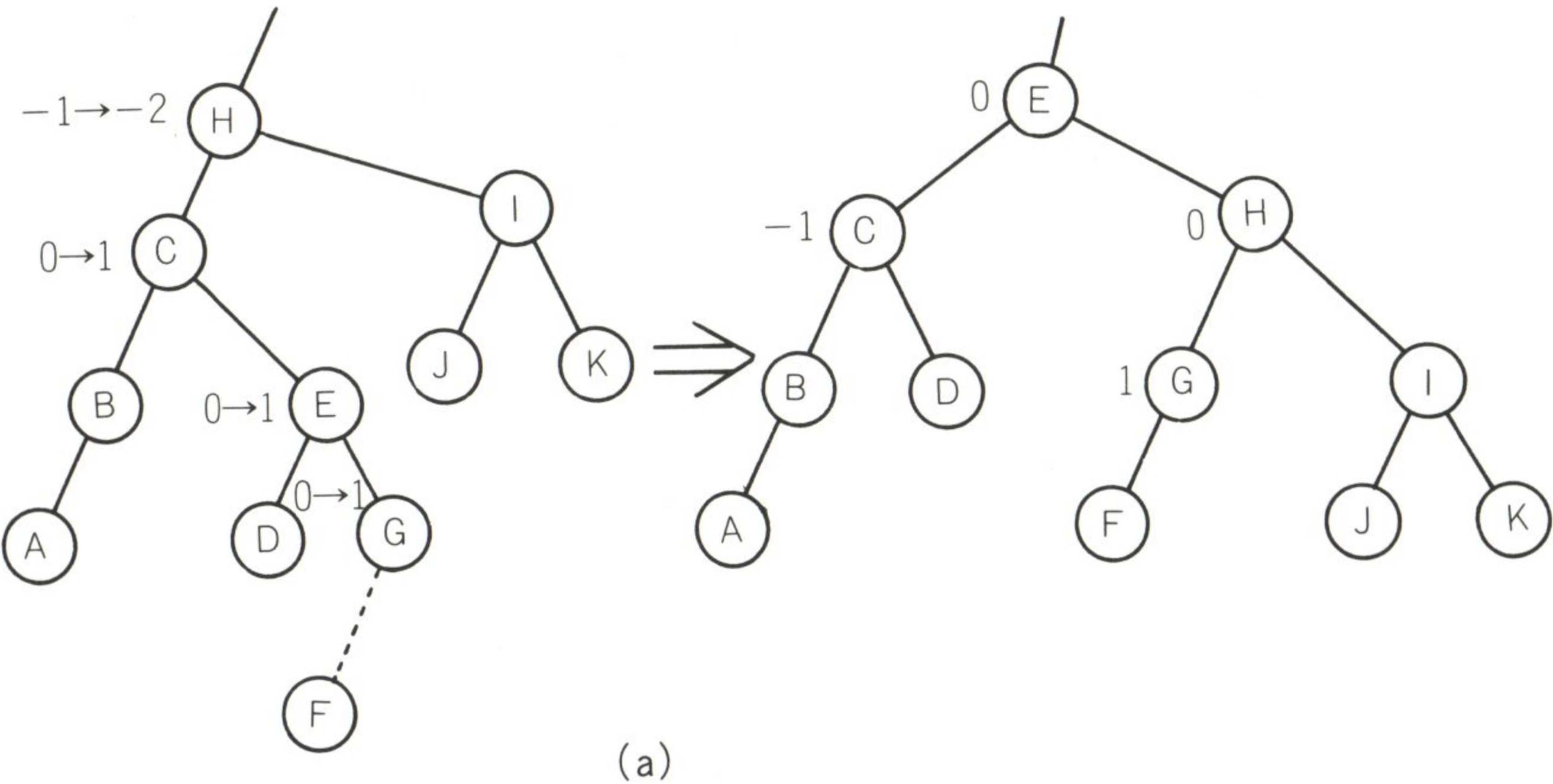
はじめに，通常の２分木の削除の手順に従って節を削除します．そして，実際に情報が除かれた節を出発点としてバランスの調整を行います．考え方は，挿入の場合とよく似ています．

削除の場合は，挿入の場合とは逆に，削除される節を含む部分木の高さが減じ

●図2.5.4 2重回転の3つの型

(a)

(b)

(c)

ます．したがって，その親の節のバランスは次のようになります．

●削除による親のバランスの変化

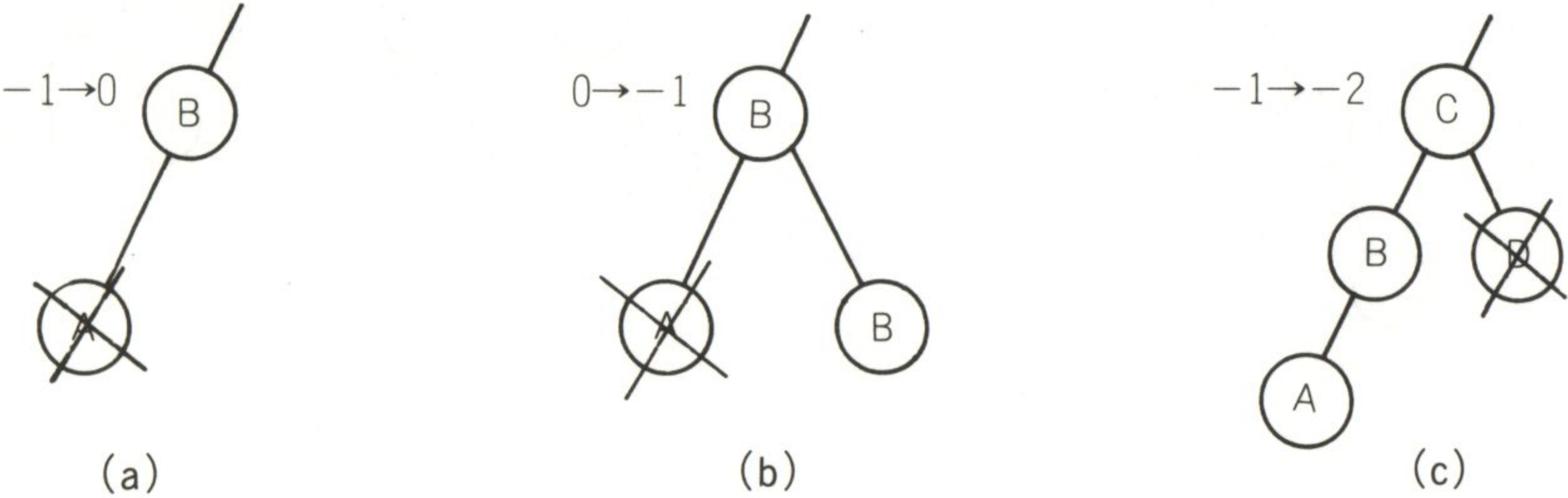
削除前のバランス	削除後のバランス 左の部分木	削除後のバランス 右の部分木
−1	0	−2
0	1	−1
1	2	0

その1 削除された結果，その親のバランスが0，すなわちバランスした場合には，その部分木の高さが1減じたことになります．したがって，木のバランスについて考えれば，アンバランスとなる可能性があるので，さらにその親について調べる必要があります．

その2 結果が1または−1になった場合には，その部分木の高さは変わらないので，調整は終わりです．

その3 結果が2または−2になった場合には，回転処理が必要になります．

上の3つの場合を図2.5.5の(a)，(b)，(c)に例示します．×印が削除される節を表します．バランスの変化を挿入の場合にならって−1→0のように表します．

●図2.5.5　削除によるバランスの変化

もう少し具体的に削除の手順をまとめると，次のようになります．

STEP1　削除すべき節を発見する．

STEP2　削除する節の左部分木がなければ，右部分木を持ち上げる．さもなくば，左部分木の最大の節を回転する．

(STEP1，STEP 2 は，2分木の削除の手順である．STEP 3 以降にてバランスの調整を行う．)

STEP3　削除された節の親のバランスを調整する．削除の結果親がバランスしたならば，その節を含む部分木の高さが1減じたことになるので，さらにその親を調べる．さもなくば，STEP 4 へ．

STEP4　親のバランスが－1または1ならば調整は終わりである．

STEP5　親のバランスが－2または2ならば，回転処理で再調整する．以下の説明では，左の部分木から削除され，バランスが－2となったものとして説明する．

図2.5.6(a)において，節Bが削除され，その結果としてEのバランスが－1から－2になったとする．この場合，Eの右の子Pのもともとのバランスの状態によって次のように処理される((a)ではPはバランスしている)．

［ケース1］　0（バランスしている）（1重回転）

Pを部分木の根とするように持ち上げ，EをPの左の子，Pの左の子をEの右の子にする．バランスは，Pを根として(1, －1, x)となる(xはPの右部分木のもともとのバランス)．

［ケース2］　1（右に傾いている）　（1重回転）

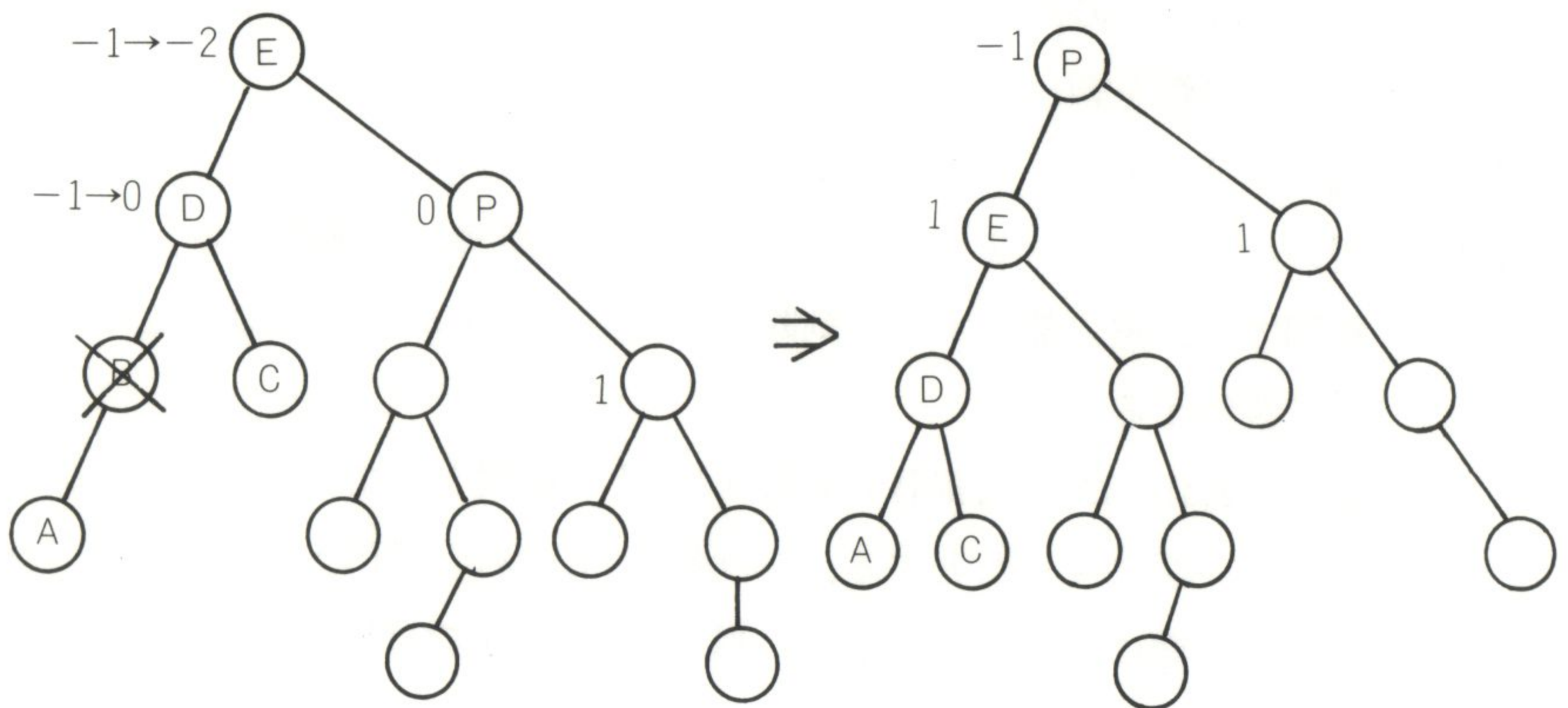
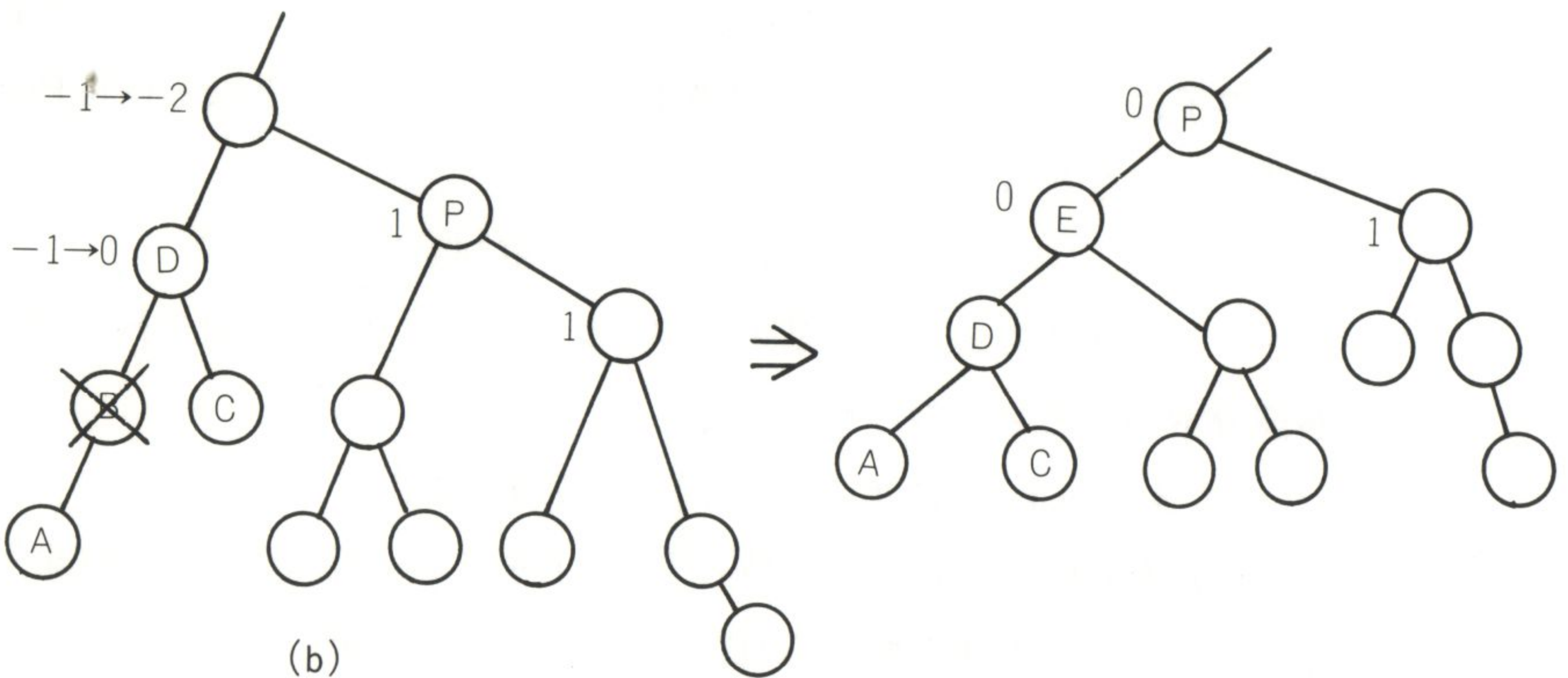
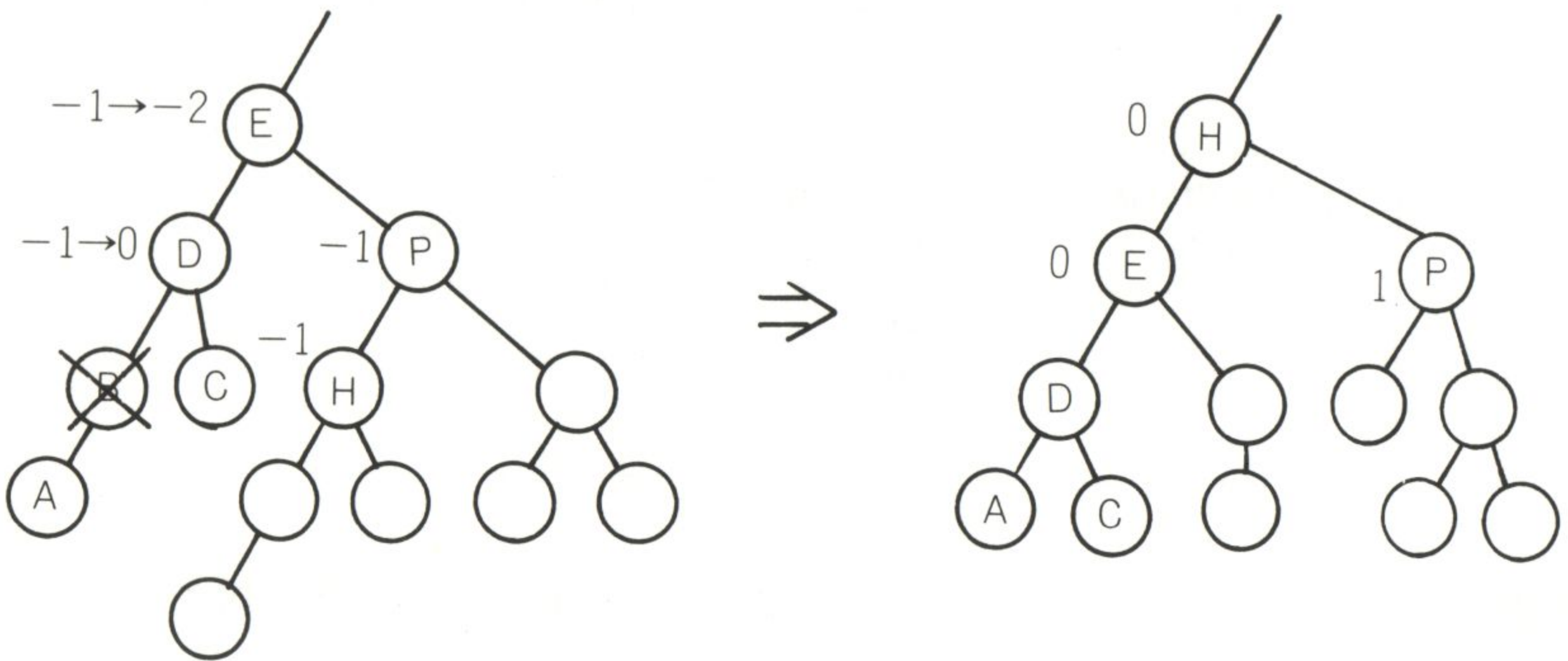
ケース1と同様であるが，結果のバランスが(0, 0, x)となる(xはPの右部分木のもともとのバランス)．

［ケース3］－1（左に傾いている）　（2重回転）

Hを部分木の根とするように持ち上げ，Eをその左部分木，Pを右部分木とするように2重回転する（図2.5.6(c)参照）．結果のバランスは，Hのバランスによって次のようになる（図(c)では，Hのバランスは－1である）．

はじめのHのバランス	結果のバランス E	H	P
1	−1	0	0
0	0	0	0
−1	0	0	1

STEP6　結果として，部分木がバランスした（＝0）ならば（図の(b)と(c)のような場合），STEP 3 へ戻る．さもなくば，終了する．

●図2.5.6 回転によるバランスの調整

(a)

(b)

(c)

2.5.4 プログラミング

AVL-木を利用した検索プログラムの要点について説明します．

データ型nodeの宣言やCOMMON宣言については，**2.3**で説明されていますので，ここでは省略します．また，未利用の要素や削除された要素を自由リストに加えておく方法については，**1.7.3**で述べましたので，そちらを参照してください．

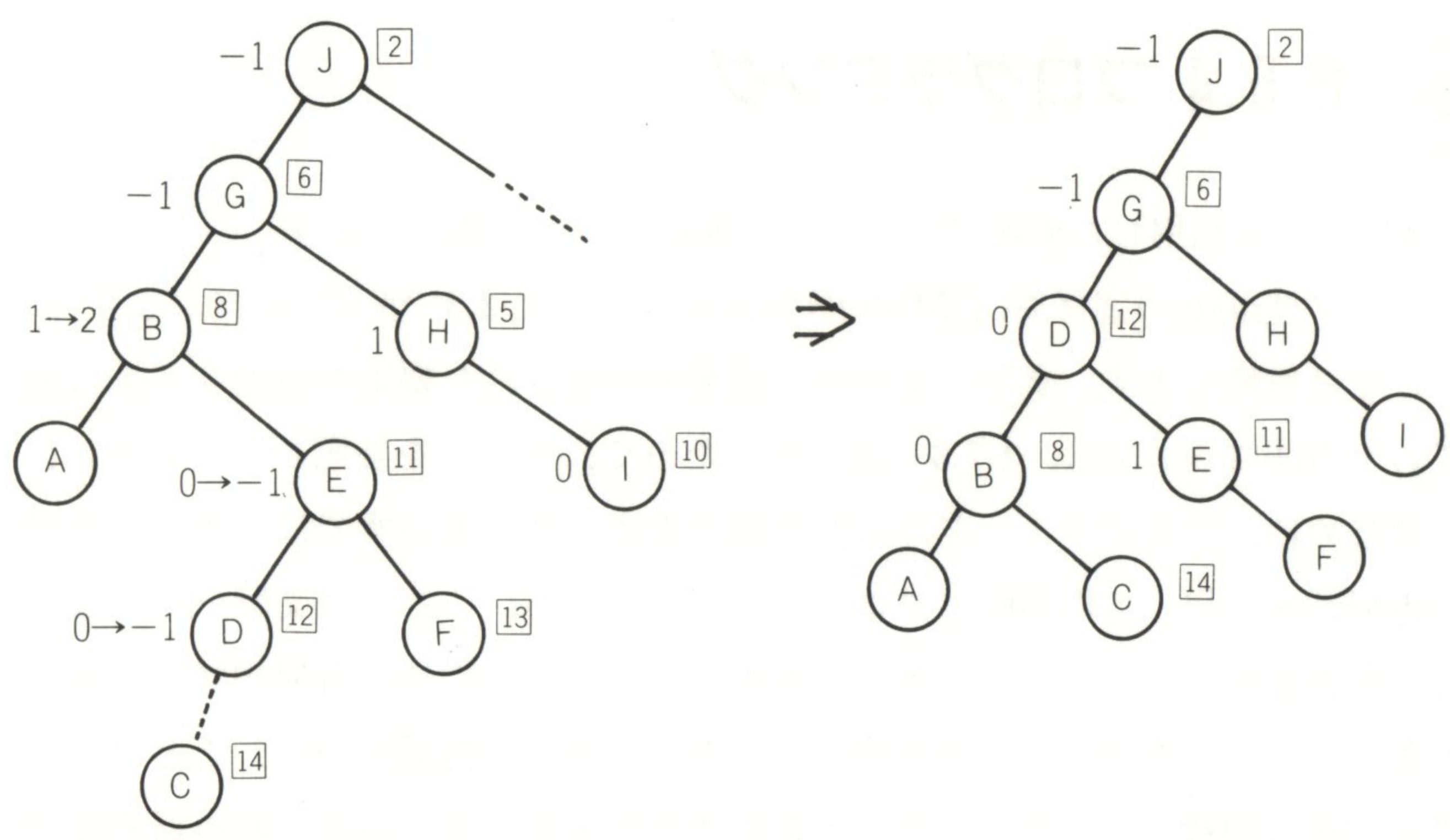
はじめに，プログラミングの方針を述べます．2分木の所では，アルゴリズムの再帰性を利用して再帰的にプログラムしましたが，ここでは文献(1)にならって，非再帰的にプログラムします．本書のプログラムでは，再帰的プログラムでも差し支えありませんが，実用的なプログラムでは，再帰呼び出しごとにスタックが大量に消費されるといったことが生ずるのを避けるために，非再帰的なプログラムとします(単語帳プログラムでは，1語分のメモリが呼び出しごとに消費され，すぐにスタックが足りなくなってしまった)．

挿入の場合でも削除の場合でも，挿入あるいは実際に削除される節がどの節の子になっているかを知る必要があります．したがって，対象となる節を捜すまでにたどった道順を，大域的な変数(SHARED宣言された変数)に保存しておく必要があります．そのための2個の配列と1個の単純変数を次のように宣言しておきます．

```
DIM SHARED path(20) AS INTEGER
DIM SHARED direction(20) AS INTEGER
DIM SHARED level
```

これらの変数の働きを調べるために，図2.5.7の挿入を例にとります．図では節Jを根とします．図の中の□で囲まれた数字は，節が格納されている配列の添字です．

はじめに，挿入すべき情報“C”が格納される位置を捜します．プログラム中では，関数searchを用います．図から直ちにJの左の部分木，Gの左の部分木，……とたどっていって，Dの左の子の位置におかれることがわかります．自由リストから添字14の配列要素が利用できるとすると，path( )，direction( )，levelは，次の

●図2.5.7 挿入の例．Ⓙが根，Ⓒが挿入される

ような値をもっています．

level	path	direction
1	2	−1
2	6	−1
3	8	1
4	11	−1
5	12	−1
6	14	0

変数levelは，根から数えて6番目の節に情報“C”が格納されるので，最終的には6の値を持っています．path( )は各節の配列添字，direction( )は節をたどる方向で，−1は左，1は右，0はそこで終わりを表します．

2分木の場合には，Dの左にCを接続して終了ですが，AVL-木では，バランスの調整が必要です．この例は，図2.5.4(b)に同じ型なので，2重回転が生じます．2重回転に至るまでの手順は次のようになります．

挿入された節Cの親はDです．それは，path(5)により知られます．プログラムふうに書けば，

```
level = level-1        (level = 5)
if x(path(level)).weight = balanced then
        x(path(level)).weight = direction(level)
end if
```

として，Dの調整をします．ここに，balancedは記号定数で0とします．Dはバランスしていて，その左にCが接続されました．“その左”という情報は，direction(5)に入っています．同様にして，Eのバランスも－1になります．Bについては，

```
level = level-1      (level = 3)
if x(path(level)).weight = direction(level) then
       ......
       call doublerotation(level,p%)
       ......
       x(path(level)).weight = -direction(level)
       x(path(level+1)).weight = balanced
       ......
end if
```

となります．節Bは，バランスが1，すなわち右に傾いており，その右の部分木に節Cが挿入されたので，傾きが2になってしまいます．プログラム上は，2にしないで2重回転をさせます．

path( )とdirection( )は，このようにして挿入された節の親を順にたどるために利用されます．この考えは，削除のプログラムでも同様です．

level＝3の場合に，実際に2重回転される部分を追跡してみます．プロシージャdoublerotationの内部を順に分解して説明します(リスト2.5.1参照のこと)．そのようすを図2.5.8の(a)～(d)に示します．◎の節がpath( )で指されているものです．

はじめにプロシージャConnectNode(3-1，path(3+2)，p%)が実行されます．path(5)は節Dを指します．Dをlevel＝2，すなわち節Gの左の子とします(図2.5.8(b))．

次に，プロシージャExchangeNode(3，5)により，path(5)の子をpath(3)の子にし，path(3)をpath(5)の子にします．左右の別は，direction(3)に従います(同

●図2.5.8

(a)

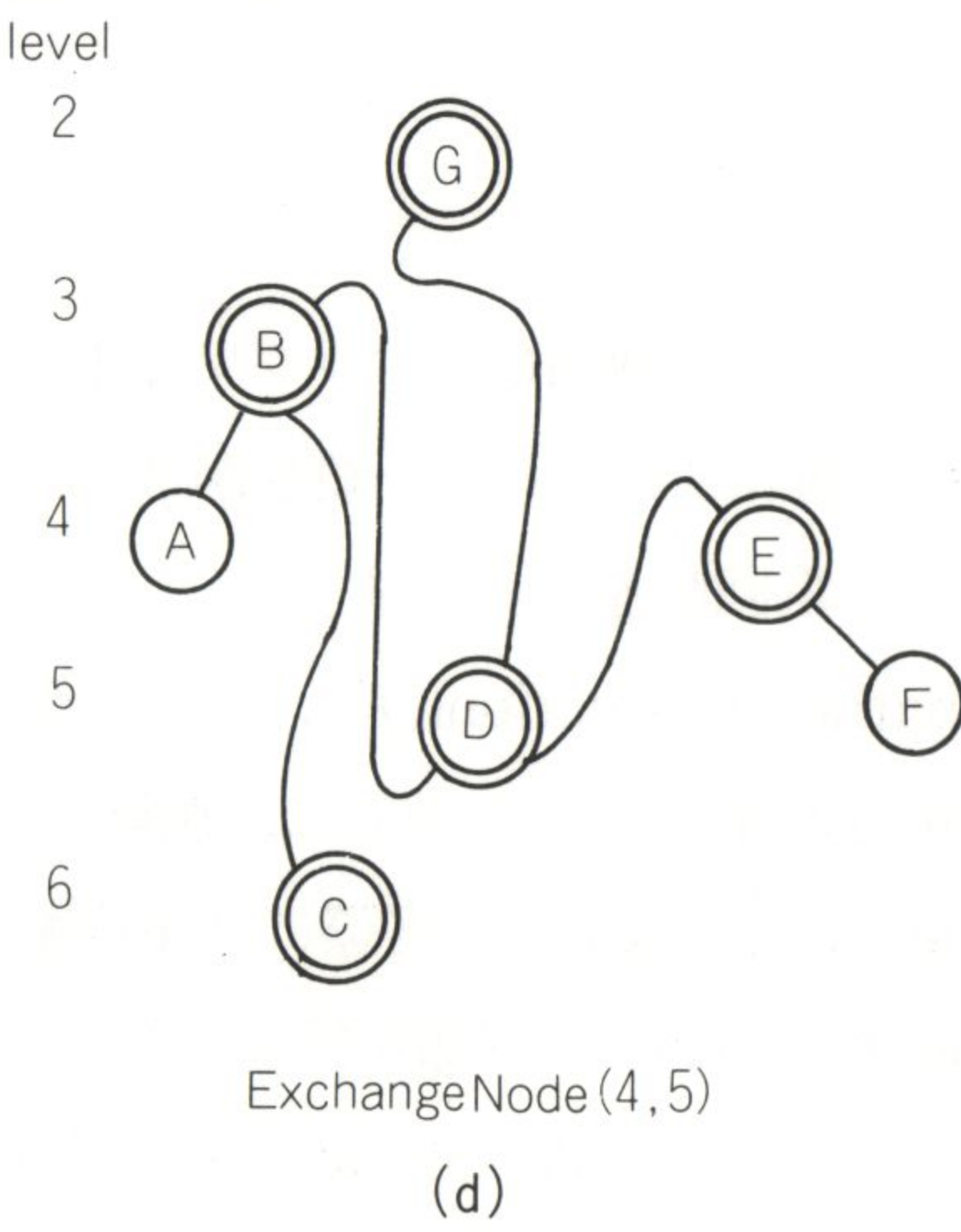
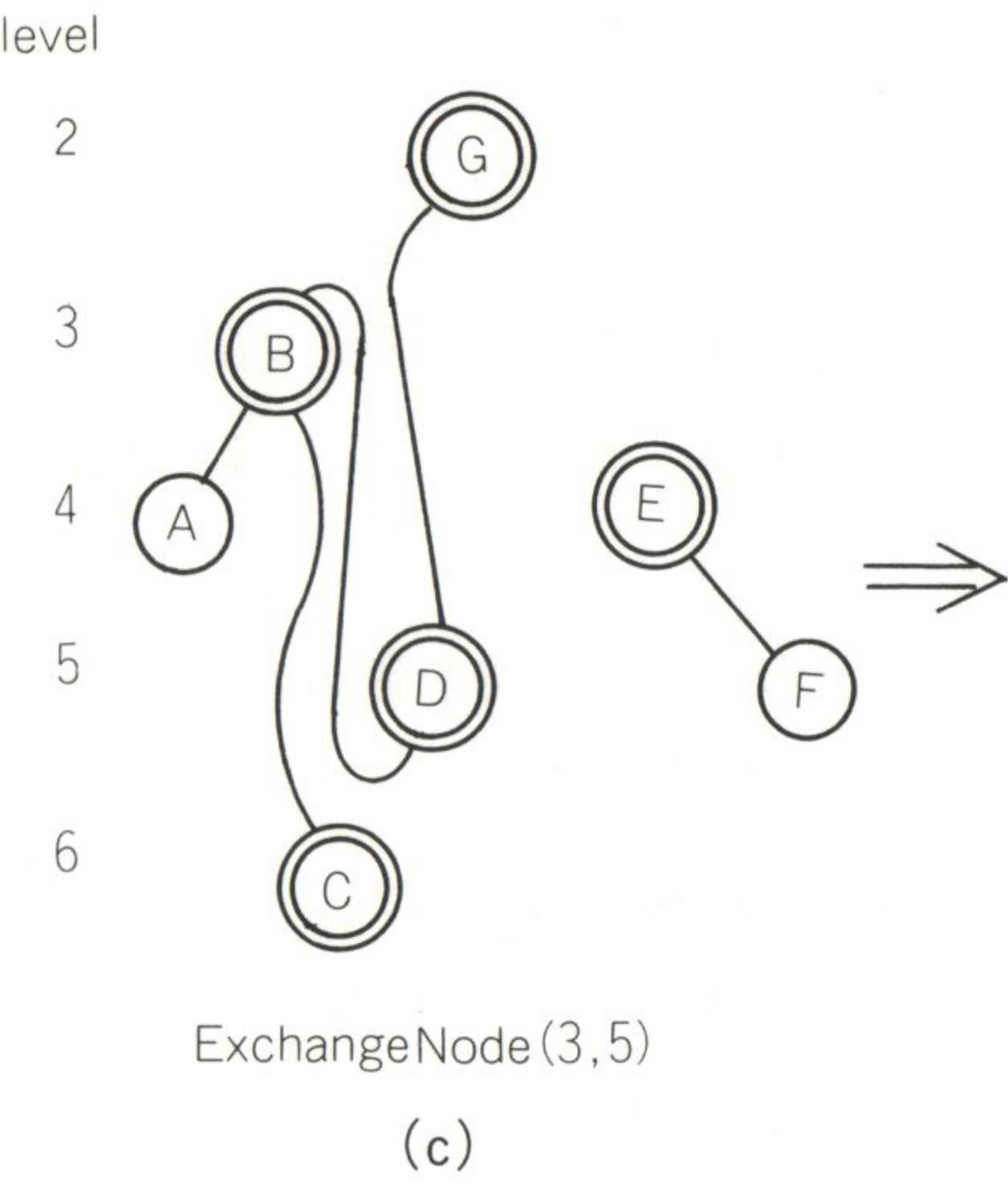
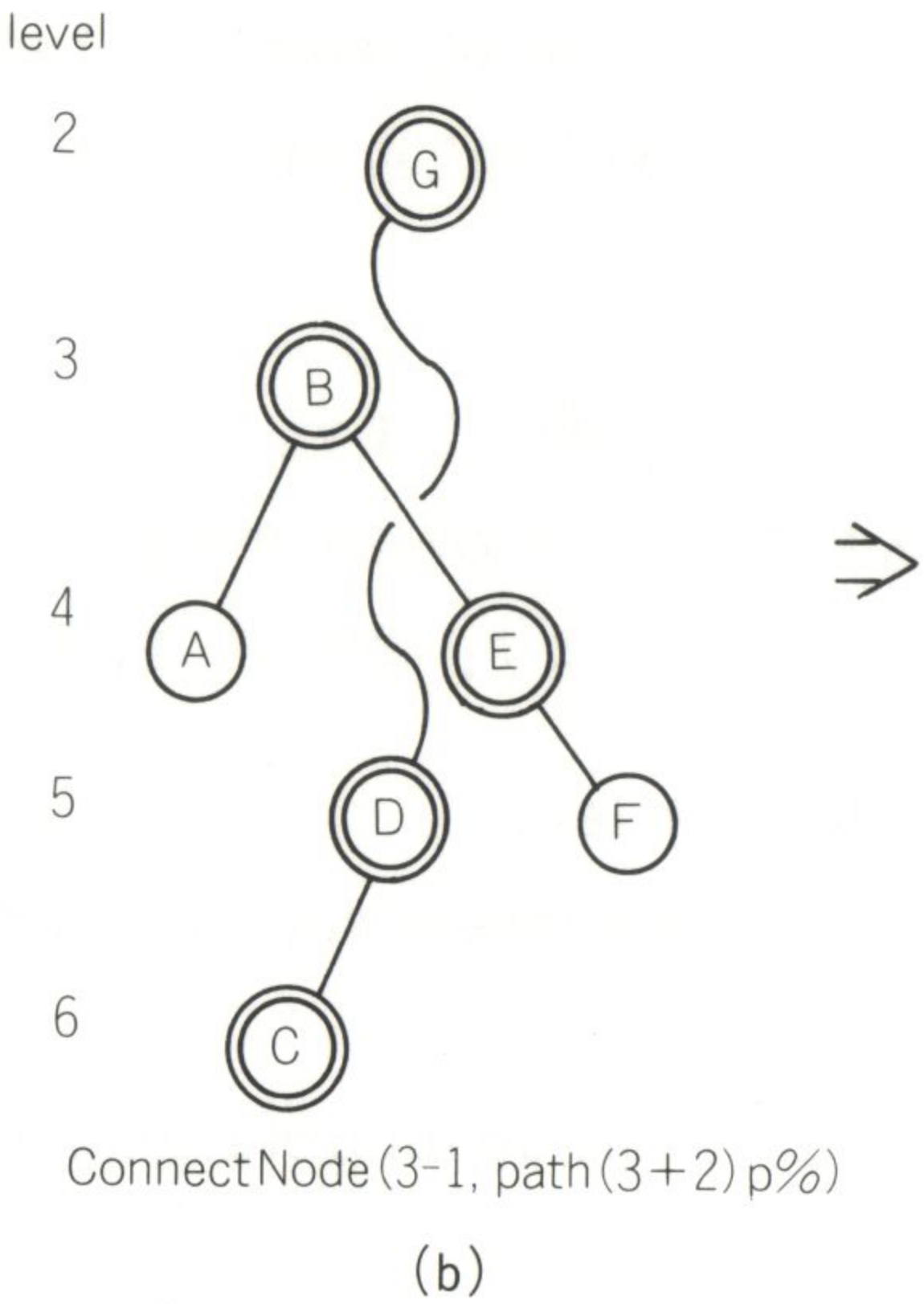
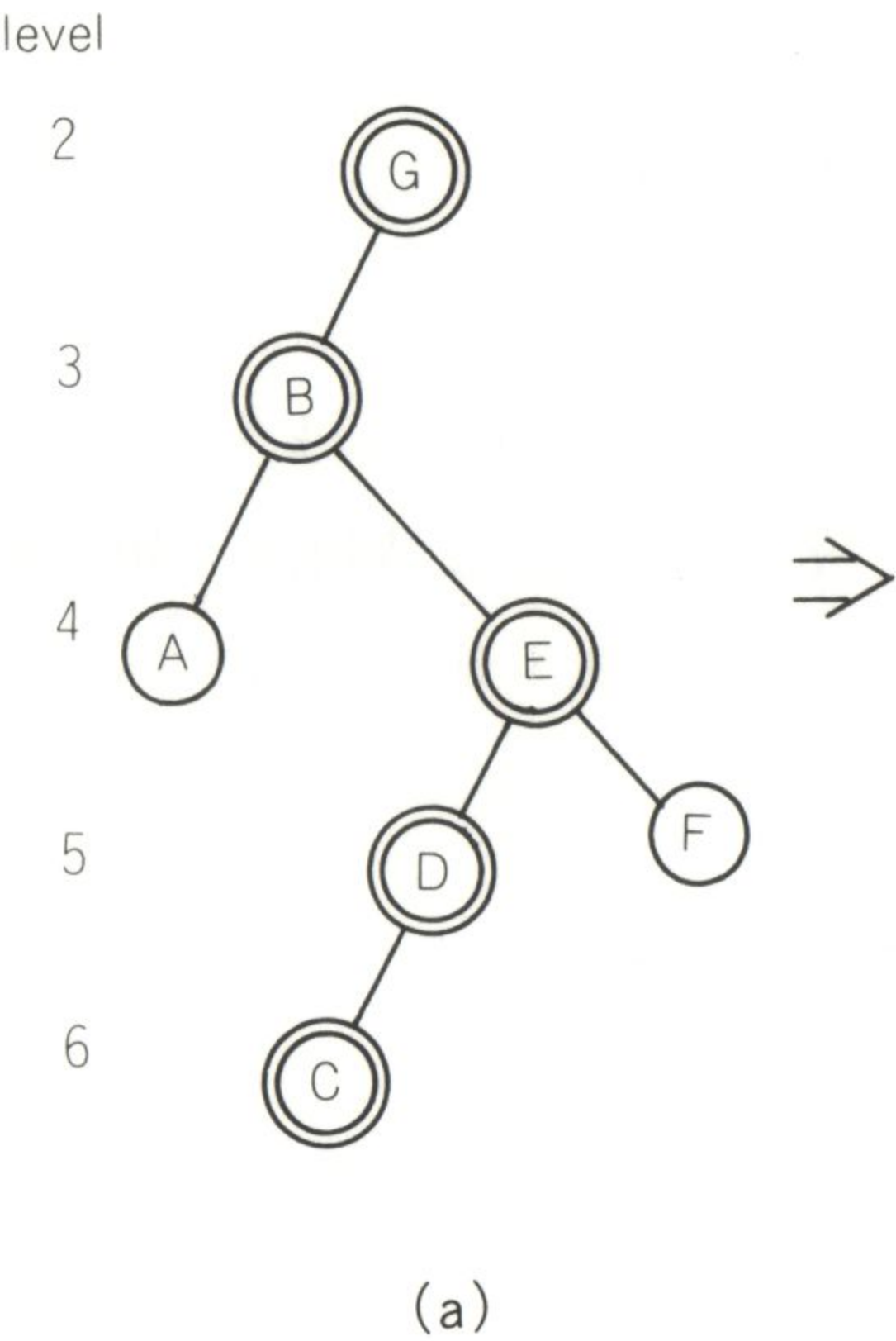
ConnectNode(3-1, path(3+2)p%)

(b)

ExchangeNode(3,5)

(c)

ExchangeNode(4,5)

(d)

図(c)).

最後にプロシージャExchangeNode(4,5)により,path(5)のもう一方の子をpath(4)の子にし，path(4)をpath(5)の子にします．左右の別は，direction(4)に従います(同図(d))．(注：図(d)では，path(5)のもう一方の子はないので，描かれていません.)

配列path( )とdirection( )の大きさは,木の節の数をNとすると，4 ×$\log_2$N程度あれば充分です.

プログラムリストを**リスト2.5.1**に示します．これは，**2.3**のサブモジュールとして使用されるものなので,AVL-木の処理以外にPrintTreeとかDispTreeが含まれています．それらは，基本的には，**1.7**の２分木のそれと同様のアルゴリズムを使用しています．したがって，解説は省略しました.

●リスト 2.5.1 AVL-木処理モジュール

```
 1: DECLARE FUNCTION IsKanji! (c$)
 2: DECLARE SUB AddNodeToFreeList (where%)
 3: DECLARE SUB AddPath (DirNow!)
 4: DECLARE SUB AdjustBalances (p%)
 5: DECLARE FUNCTION MaxNode! ()
 6: DECLARE SUB DoubleRotation (l!, p%)
 7: DECLARE SUB InitFreeList ()
 8: DECLARE SUB Delete (s$, p%, f%)
 9: DECLARE FUNCTION GetNode! ()
10: DECLARE SUB Insert (s$, b%, c$, p%)
11: DECLARE SUB PrintTree (p%, depth!, u$, l$)
12: DECLARE FUNCTION Search! (item$, p%)
13: DECLARE SUB ConnectNode (level%, NodeNo%, p%)
14: DECLARE SUB ExchangeNode (a, b)
15: DECLARE SUB Rotation (l, p%)
16:
17: CONST InfSize = 20
18: CONST SynopSize = 70
19: CONST ListSize = 100
20: CONST balanced = 0, left = -1, right = 1, Nil = 0
21: CONST false = 0, true = NOT false
22: CONST PCol = 30
23: CONST DCol = 52
24:
25: TYPE Node
26:   inf AS STRING * InfSize
27:   acc AS INTEGER
28:   synop AS STRING * SynopSize
29:   left AS INTEGER
30:   right AS INTEGER
31:   weight AS INTEGER
32: END TYPE
33:
34: DIM SHARED path(20)  AS INTEGER
35: DIM SHARED direction(20)  AS INTEGER
36: DIM x(ListSize)    AS Node
37: DIM SHARED level
38:
39: COMMON SHARED x()   AS Node
40:
41: DEFINT W
```

```
 42: SUB AddNodeToFreeList (where)
 43:     x(where).right = x(0).right
 44:     x(0).right = where
 45: END SUB
 46:
 47: SUB AddPath (DirNow)
 48:     direction(level) = DirNow
 49:     level = level + 1
 50:     IF DirNow = left THEN
 51:         path(level) = x(path(level - 1)).left
 52:     ELSE
 53:         path(level) = x(path(level - 1)).right
 54:     END IF
 55:     direction(level) = 0
 56: END SUB
 57:
 58: SUB AdjustBalances (p%)
 59: again:
 60: '=== Step 3 ===
 61:     FOR level = level - 1 TO 1 STEP -1
 62:         PathNow = path(level)
 63:         SELECT CASE x(PathNow).weight
 64:             CASE balanced
 65:                 IF direction(level) = left THEN
 66:                     x(PathNow).weight = right
 67:                 ELSE
 68:                     x(PathNow).weight = left
 69:                 END IF
 70:             CASE left
 71:                 IF direction(level) = left THEN
 72:                     x(PathNow).weight = balanced
 73:                 ELSE
 74:                     x(PathNow).weight = left + left
 75:                 END IF
 76:             CASE right
 77:                 IF direction(level) = left THEN
 78:                     x(PathNow).weight = right + right
 79:                 ELSE
 80:                     x(PathNow).weight = balanced
 81:                 END IF
 82:         END SELECT
 83:         IF x(PathNow).weight <> balanced THEN
 84:             EXIT FOR
 85:         END IF
 86:     NEXT level
 87:
 88: '=== Step 4 === Rebalancing
 89:     LevelNow = level
 90:     DirNow = direction(LevelNow)
 91:     WeightNow = x(PathNow).weight
 92:
 93:     IF WeightNow = left OR WeightNow = right OR level <= 0 THEN
 94:         EXIT SUB
 95:     END IF
 96:     '  Now, tree is unbalanced.
 97:     'search target node
 98:
 99: '=== Step 5 ===
100:     CALL AddPath(-DirNow)
101:
102:     TargetWeight = x(path(level)).weight
103:     SELECT CASE TargetWeight
104:         CASE balanced           ' rotation (case 1), Fig.2.5.6(a)
105:             CALL Rotation(level - 1, p%)
106:             x(path(level)).weight = DirNow
107:             x(path(level - 1)).weight = -DirNow
108:         CASE -DirNow            ' rotation (case 2), Fig.2.5.6(b)
109:             CALL Rotation(level - 1, p%)
110:             x(path(level)).weight = balanced
```

→See page 203 (Step 3)
→See page 203 (Step 4)
→See page 203 (Step 5)

```
111:                x(path(level - 1)).weight = balanced
112:
113:            CASE DirNow       ' double rotations (case 3)
114:                CALL AddPath(DirNow)        'rotated node
115:                TargetWeight = x(path(level)).weight
116:                CALL DoubleRotation(level - 2, p%)
117:
118:                x(path(level)).weight = balanced
119:                SELECT CASE TargetWeight
120:                    CASE DirNow
121:                        x(path(level - 1)).weight = -DirNow
122:                        x(path(level - 2)).weight = balanced
123:                    CASE -DirNow
124:                        x(path(level - 1)).weight = balanced
125:                        x(path(level - 2)).weight = DirNow
126:                    CASE balanced                         ' Fig.2.5.6(c)
127:                        x(path(level - 1)).weight = balanced
128:                        x(path(level - 2)).weight = balanced
129:                END SELECT
130:        END SELECT
131:
132: '=== Step 6 ===                                            →See page 203
133:        IF x(path(LevelNow)).weight = balanced AND LevelNow > 1 THEN
134:            PRINT "LevelNow="; LevelNow
135:            level = LevelNow
136:            GOTO again
137:        END IF
138: END SUB
139:
140: DEFSNG W
141: SUB ConnectNode (l%, NodeNo%, p%) 'Connect NodeNo'th node to path(l)'th node
142:        IF l% = 0 THEN
143:            p% = NodeNo%
144:        ELSE
145:            IF direction(l%) = left THEN
146:                x(path(l%)).left = NodeNo%
147:            ELSE
148:                x(path(l%)).right = NodeNo%
149:            END IF
150:        END IF
151: END SUB
152:
153: '
154: SUB Delete (t$, p%, f%)
155:        DEFINT W
156:        f% = true
157:        s$ = t$ + SPACE$(InfSize - LEN(t$))
158:        IF p% = 0 THEN
159:            f% = false
160:            EXIT SUB
161:        END IF
162:
163: '=== Step 1 ===                                            →See page 202
164:        DeletedNOde% = Search(s$, p%)
165:        DeletedLevel = level
166:        IF DeletedNOde% = 0 THEN
167:            f% = false
168:            EXIT SUB
169:        END IF
170:
171: '=== Step 2 ===                                            →See page 202
172:        IF x(DeletedNOde%).left = Nil THEN
173:            CALL ConnectNode(DeletedLevel - 1, x(DeletedNOde%).right, p%)
174:            x(x(DeletedNOde%).right).weight = x(DeletedNOde%).weight
175:        ELSE ' search MaxNode in left-tree
176:            where = MaxNode
177:            CALL ConnectNode(level - 1, x(where).left, p%)
178:            x(where).left = x(DeletedNOde%).left
179:            x(where).right = x(DeletedNOde%).right
```

```
180:         CALL ConnectNode(DeletedLevel - 1, where, p%)
181:         path(DeletedLevel) = where
182:         x(where).weight = x(DeletedNOde%).weight
183:     END IF
184:
185:     CALL AddNodeToFreeList(DeletedNOde%)
186: '=== Step 3...6 ===
187:     CALL AdjustBalances(p%)
188:
189: END SUB
190:
191: DEFSNG W
192: SUB DispTree (p%, depth, u$, l$)
193:
194: DIM Spell(1 TO InfSize) AS STRING * 1
195: DIM c AS STRING * 1
196:
197:     IF p% <> 0 THEN
198:
199:         CALL DispTree(x(p%).left, depth + 1, u$, l$)
200:
201:
202:         word$ = x(p%).inf
203:         Head$ = UCASE$(LEFT$(word$, 1))
204:         Accnt = x(p%).acc
205:
206:         IF Head$ >= u$ AND Head$ <= l$ THEN
207:
208:             FOR i = 1 TO InfSize
209:                 Spell(i) = MID$(word$, i, 1)
210:             NEXT i
211:
212:             PRINT SPC(5);
213:             FOR i = 1 TO InfSize
214:                 IF Accnt = i THEN
215:                     COLOR 4
216:                     PRINT Spell(i);
217:                     COLOR 7
218:                 ELSE
219:                     PRINT Spell(i);
220:                 END IF
221:             NEXT i
222:
223:             IF LEN(RTRIM$(x(p%).synop)) > DCol THEN
224:
225:                 c = MID$(x(p%).synop, DCol, 1)
226:
227:                 IF IsKanji(c) THEN
228:
229:                     PRINT USING ": @"; LEFT$(x(p%).synop, DCol - 2)
230:                     PRINT SPC(25); : PRINT USING "  @"; MID$(x(p%).synop, DCol - 1)
231:                 ELSE
232:                     PRINT USING ": @"; LEFT$(x(p%).synop, DCol)
233:                     PRINT SPC(25); : PRINT USING "  @"; MID$(x(p%).synop, DCol + 1)
234:                 END IF
235:             ELSE
236:                 PRINT USING ": @"; x(p%).synop
237:             END IF
238:
239:         END IF
240:
241:             CALL DispTree(x(p%).right, depth + 1, u$, l$)
242:     END IF
243:
244: END SUB
245:
246: SUB DoubleRotation (l, p%)
247:
248:     CALL ConnectNode(l - 1, path(l + 2), p%) 'Connect path(l+2) to path(l-1)
```

```
249:     CALL ExchangeNode(l, l + 2)      'Connect path(l) to path(l+2)
250:     CALL ExchangeNode(l + 1, l + 2) '        path(l+1) to path(l+2)
251:
252: END SUB
253:
254: SUB ExchangeNode (a, b)
255:
256:     IF direction(a) = left THEN
257:         x(path(a)).left = x(path(b)).right
258:         x(path(b)).right = path(a)
259:     ELSE
260:         x(path(a)).right = x(path(b)).left
261:         x(path(b)).left = path(a)
262:     END IF
263: END SUB
264:
265: FUNCTION GetNode
266:
267:     GetNode = x(0).right
268:     x(0).right = x(x(0).right).right
269:
270: END FUNCTION
271:
272: SUB InitFreeList
273:     FOR i = 1 TO ListSize
274:         x(i - 1).right = i
275:     NEXT i
276:     x(ListSize).right = 0
277:
278: END SUB
279:
280: SUB Insert (a$, b%, c$, p%)
281:     s$ = a$ + SPACE$(InfSize - LEN(a$))
282:     where = Search(s$, p%)
283:     IF where <> 0 THEN
284:         x(where).acc = b%
285:         x(where).synop = c$
286:         EXIT SUB
287:     END IF
288:     where = GetNode
289:     IF where = 0 THEN
290:         PRINT "No room!! Aborted."
291:         EXIT SUB
292:     END IF
293: '     Add a new Node to the lowest level
294:     x(where).inf = s$
295:     x(where).acc = b%
296:     x(where).synop = c$
297:     x(where).left = Nil
298:     x(where).right = Nil
299:     x(where).weight = balanced
300:     level = level + 1
301:     path(level) = where
302:     direction(level) = balanced
303:
304: '     Connect path(level) to path(level-1)
305:     CALL ConnectNode(level - 1, path(level), p%)
306:
307:     FOR level = level - 1 TO 1 STEP -1
308:         weight = x(path(level)).weight
309:         IF weight = direction(level) THEN
310:             IF direction(level) = direction(level + 1) THEN
311:                 CALL Rotation(level, p%)  ' Fig.2.5.3, Outside
312:                 x(path(level)).weight = balanced
313:                 x(path(level + 1)).weight = balanced
314:             ELSE
315:                 CALL DoubleRotation(level, p%) ' Fig.2.5.4(a)
316:                 x(path(level + 2)).weight = balanced
317:                 IF direction(level + 1) = direction(level + 2) THEN
```

```
318:                    x(path(level)).weight = balanced
319:                    x(path(level + 1)).weight = -direction(level + 1)
320:                ELSEIF direction(level + 2) <> balanced THEN
321:                    x(path(level)).weight = -direction(level) ' Fig.2.5.4(b)
322:                    x(path(level + 1)).weight = balanced
323:                ELSE
324:                    x(path(level)).weight = balanced          ' Fig.2.5.4(c)
325:                    x(path(level + 1)).weight = balanced
326:                END IF
327:
328:            END IF
329:            EXIT SUB
330:        END IF
331:        IF weight = balanced THEN
332:            x(path(level)).weight = direction(level)  'balanced -> right/left
333:        ELSE
334:            x(path(level)).weight = balanced              ' Fig.2.5.1
335:            EXIT SUB
336:        END IF
337:    NEXT level
338:
339: END SUB
340:
341: FUNCTION MaxNode
342:     where = x(path(level)).left
343:     DO WHILE where <> 0
344:         level = level + 1
345:         path(level) = where
346:         direction(level) = right
347:         where = x(where).right
348:     LOOP
349:     MaxNode = path(level)
350: END FUNCTION
351:
352: SUB PrintTree (p%, depth, u$, l$)
353:
354: DIM Spell(1 TO InfSize) AS STRING * 1
355: DIM c AS STRING * 1
356:
357:
358:     IF p% <> 0 THEN
359:
360:         CALL PrintTree(x(p%).left, depth + 1, u$, l$)
361:
362:
363:         word$ = x(p%).inf
364:         Head$ = UCASE$(LEFT$(word$, 1))
365:         Accnt = x(p%).acc
366:
367:         IF Head$ >= u$ AND Head$ <= l$ THEN
368:
369:             FOR i = 1 TO InfSize
370:                 Spell(i) = MID$(word$, i, 1)
371:             NEXT i
372:
373:             LPRINT SPC(5);
374:             FOR i = 1 TO InfSize
375:                 IF Accnt = i THEN
376:                     LPRINT CHR$(27) + "X"; Spell(i); CHR$(27) + "Y";
377:                 ELSE
378:                     LPRINT Spell(i);
379:                 END IF
380:             NEXT i
381:
382:             IF LEN(RTRIM$(x(p%).synop)) > PCol THEN
383:
384:                 c = MID$(x(p%).synop, PCol, 1)
385:                 IF IsKanji(c) THEN
386:                     LPRINT USING ": @"; LEFT$(x(p%).synop, PCol - 2)
```

```
387:                 LPRINT SPC(25); : LPRINT USING "  @"; MID$(x(p%).synop, PCol - 1)
388:             ELSE
389:                 LPRINT USING ": @"; LEFT$(x(p%).synop, PCol)
390:                 LPRINT SPC(25); : LPRINT USING "  @"; MID$(x(p%).synop, PCol + 1)
391:             END IF
392:         ELSE
393:             LPRINT USING ": @"; x(p%).synop
394:         END IF
395:
396:     END IF
397:         CALL PrintTree(x(p%).right, depth + 1, u$, l$)
398:     END IF
399:
400: END SUB
401:
402: SUB Rotation (l, p%)
403:
404:     CALL ConnectNode(l - 1, path(l + 1), p%)'Connect path(l+1) to path(l-1)
405:     CALL ExchangeNode(l, l + 1)      'Connect path(l) to path(l+1)
406:
407: END SUB
408:
409: FUNCTION Search (item$, p%)
410:     where = p%
411:     level = 0
412:     DO WHILE where <> 0
413:         IF x(where).inf >= item$ THEN
414:             dir = left
415:         ELSE
416:             dir = right
417:         END IF
418:         level = level + 1
419:         path(level) = where
420:         direction(level) = dir
421:
422:         IF x(where).inf = item$ THEN
423:             EXIT DO
424:         END IF
425:         IF direction(level) = left THEN
426:             where = x(where).left
427:         ELSE
428:             where = x(where).right
429:         END IF
430:     LOOP
431:
432:     Search = where
433:
434:  END FUNCTION
435:
```

参考文献

（1） ストーン他(寺田他訳)「コンピュータとデータ構造」CQ出版社

2.6 ファジィによる自動車のスピードコントロール

ファジィとは日本語で“あいまい”とか“ぼけ”などの意味を持つ言葉です．近年，事象のあいまいさを定量的に表現し処理するための，新しい数学的概念として規定され注目されています．1965年にはじめてL.A.Zadehによって提唱されました．

最近になって，その応用がいろいろな分野で実現され，新聞や雑誌に掲載されるようになりました．ここでは，自動車のスピードコントロールをファジィ推論により実行するモデルをプログラムしてみます．それにより，ファジィが実感的に理解できると思います．また，プログラム自体もそれほど複雑ではないので，応用も容易です．

2.6.1 ファジィ集合とメンバーシップ関数

自動車を運転する場合のスピードコントロールを，ファジィの考えを用いてモデル化してみましょう．現実の車の運転においては，多くの要素がありますが，ここでは，現在の車のスピードと前の車との車間距離のみを判断材料として，アクセルを踏み込むか，ブレーキを踏むかを加減することにします．

人が，それらを判断する場合，次のようなルールを暗黙に適用していると考えられます．すなわち，

- スピードが速くて，車間が狭いならば，ブレーキを踏む
- スピードが遅くて，車間が広いならば，アクセルを踏み込む
- スピードがまあまあで，車間もそこそこを保っているならば，アクセルはそのまま保持する

細かく分ければ，もっと多くのルールが考えられますが，とりあえず，ルール

とはこんなものを言うのだと思ってください．これを数学的に表現するのが，最初の課題です．

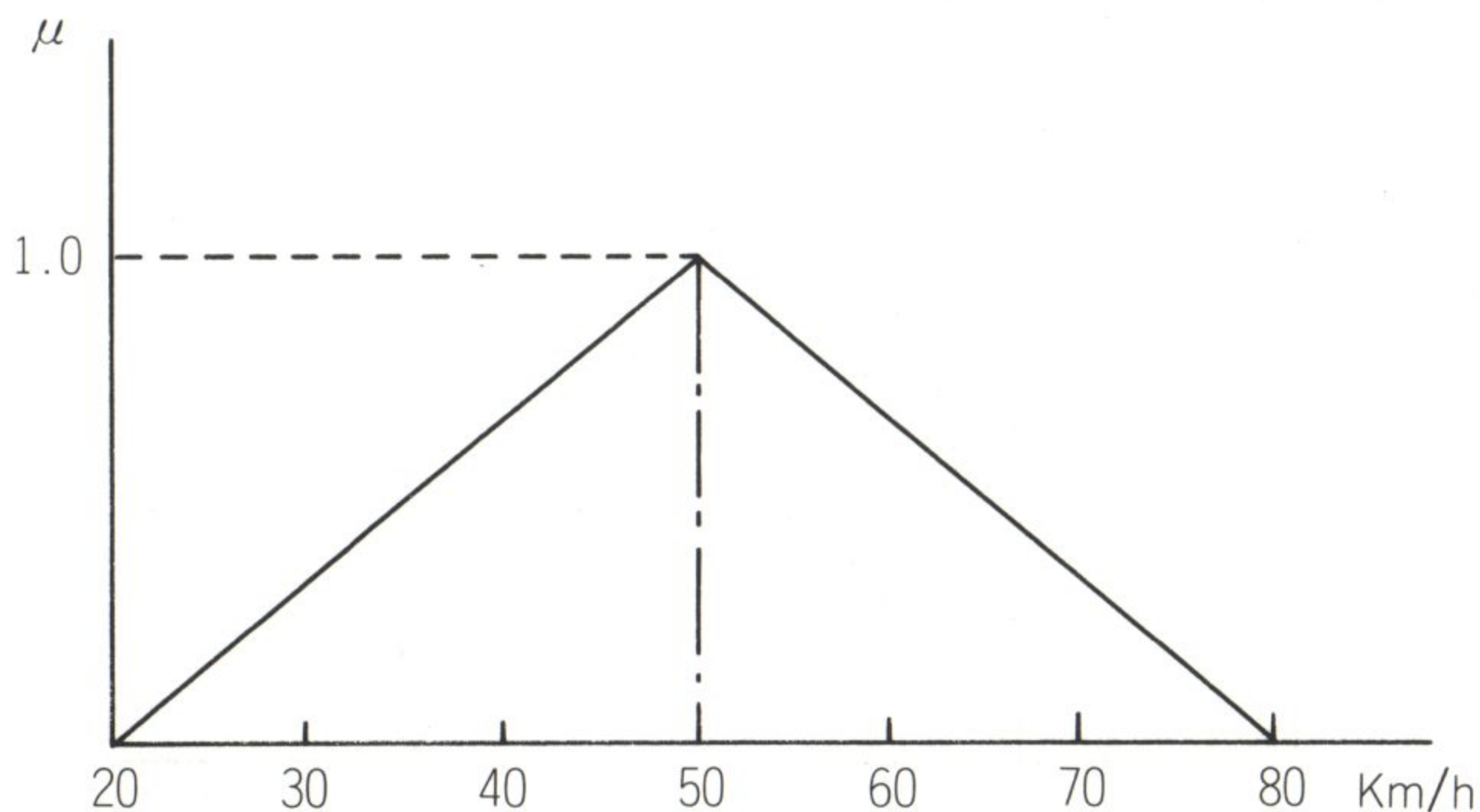
スピードが“遅い”，“まあまあ”，“速い”などを，どのようにして数学的に表すかを考えてみましょう．例として，はじめに“まあまあ”を取り上げます．

ハイウェイと一般道路では事情が異なりますが，それほど混んでいない一般道路では，“まあまあ”という感じは50Km/hと言ったところではないでしょうか．ただし，この50Km/hという数字は確定したものではなくて，周りの状況やそのときの気分によって，“まあまあ”は40Km/hになったり60Km/hになったりします．とは言っても，40Km/hよりは50Km/hの方がより“まあまあ”に近いと言ってよいでしょう．そこで，このようすを横軸にスピード，縦軸に“まあまあ”の程度をとって，グラフで表すと図2.6.1のようになります．

ここで，この図の数学的意味について考えてみます．

図では，横軸にはスピードがとられており，連続量となっています．説明のためにスピードは，10Km/h，20Km/h，…，80Km/hの離散量のみをとるものとします．

すると，“まあまあ”の程度は，50Km/hのとき値1をとり，40Km/hと60Km/hのときには2/3，30Km/hと70Km/hのときには1/3の値をとることがわかります．別の言い方をすると，“まあまあ”のスピードには，50Km/hが程度1で，40Km/hと60Km/hとが程度2/3で，30Km/hと70Km/hとが程度1/3で属していると言えます．これは，普通の集合が，ある要素が属しているか否か，すなわ

●図2.6.1

ち１か０かのどちらかであるのに対して，属している程度が１/３などと，１と０以外の値をとる点が異なります．しかし，集合と考えると，普通の集合の拡張になっていると言えるので，ファジィ集合と呼ぶことにします．言い換えると，ファジィ集合とは，“要素と，それが集合に属する程度を同時に考えにいれた集合”であると言えます．

ファジィ集合の要素がその集合に属する程度を表す関数を，メンバーシップ関数と呼びます．図2.6.1の例では，連続量で表されています．したがって，45Km/hの“らしさ”は，５/６であると読みとられます．メンバーシップ関数の値のことを，グレードと呼びます．

メンバーシップ関数の具体的な値は，なにもここにあげた三角形に限ったわけではなく，どんな値をとることもできます．モデルを作成する人や場面によって，任意の形の関数が定義されます．その自由さがファジィ集合の特徴であります．ここでは，プログラムしやすいように三角形としました．

スピードが“まあまあ”の集合のメンバーシップ関数―らしさ―は，図2.6.1で表されました．同様にして，スピードが“遅い”，“はやい”のメンバーシップ関数も表すことができます．それらをまとめて，図2.6.2に示します．

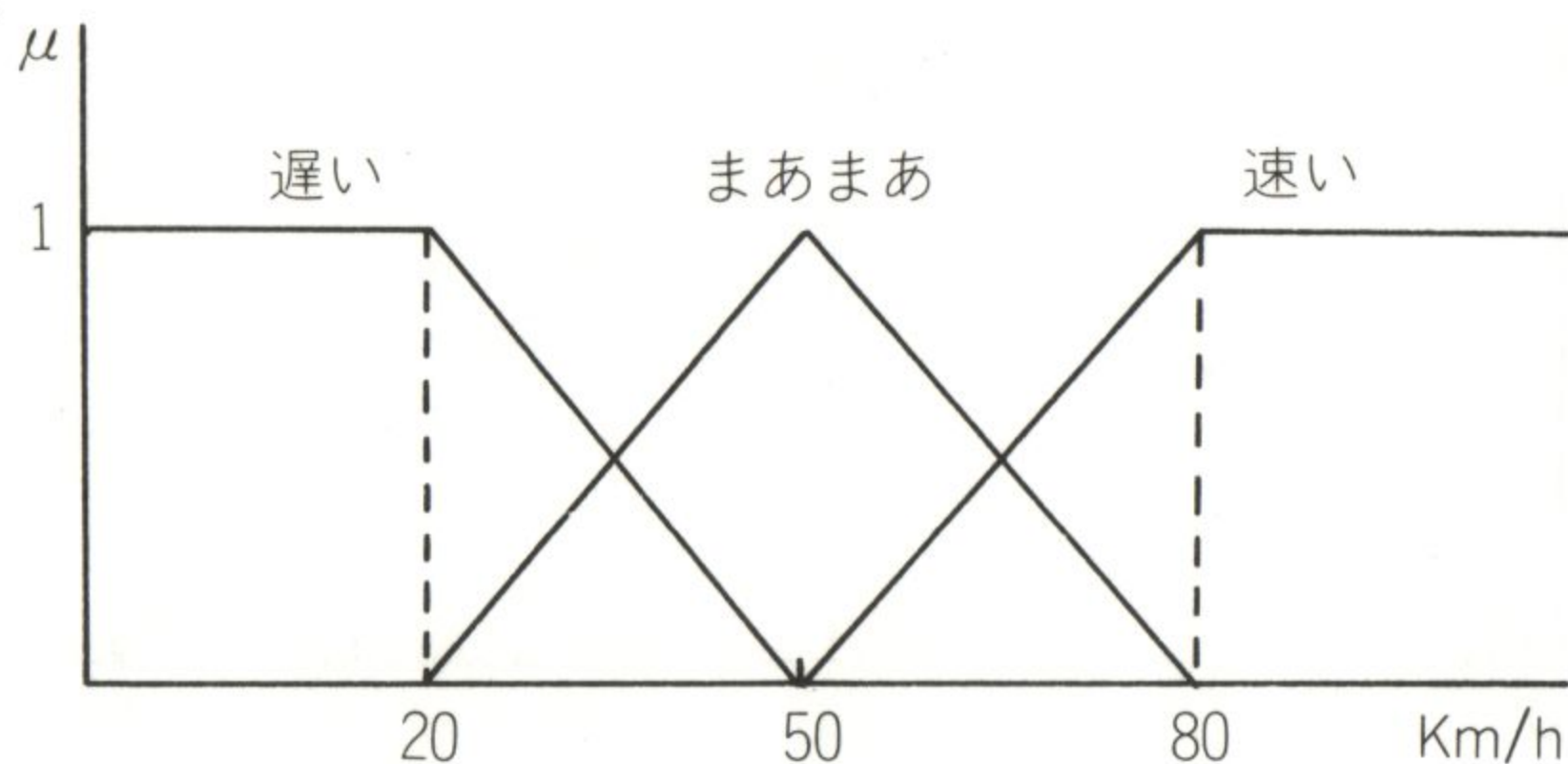
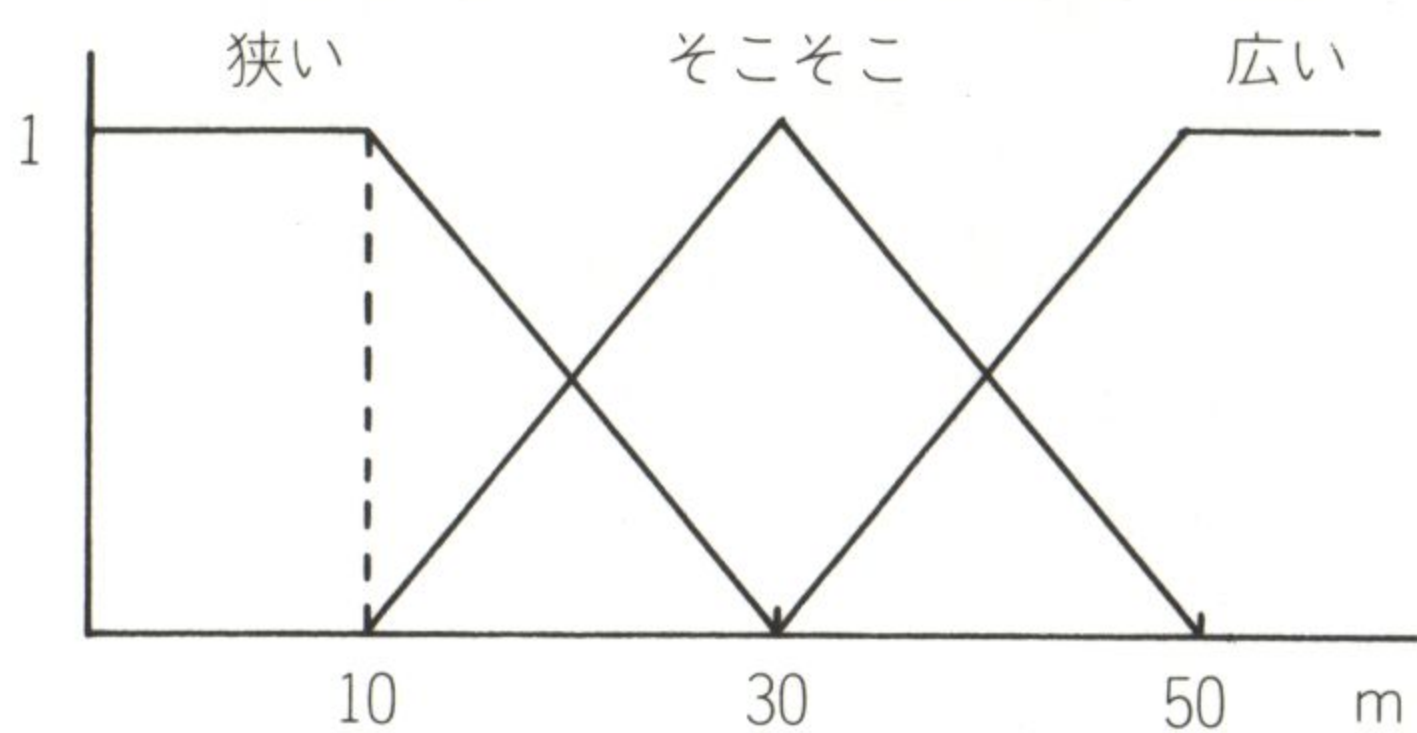
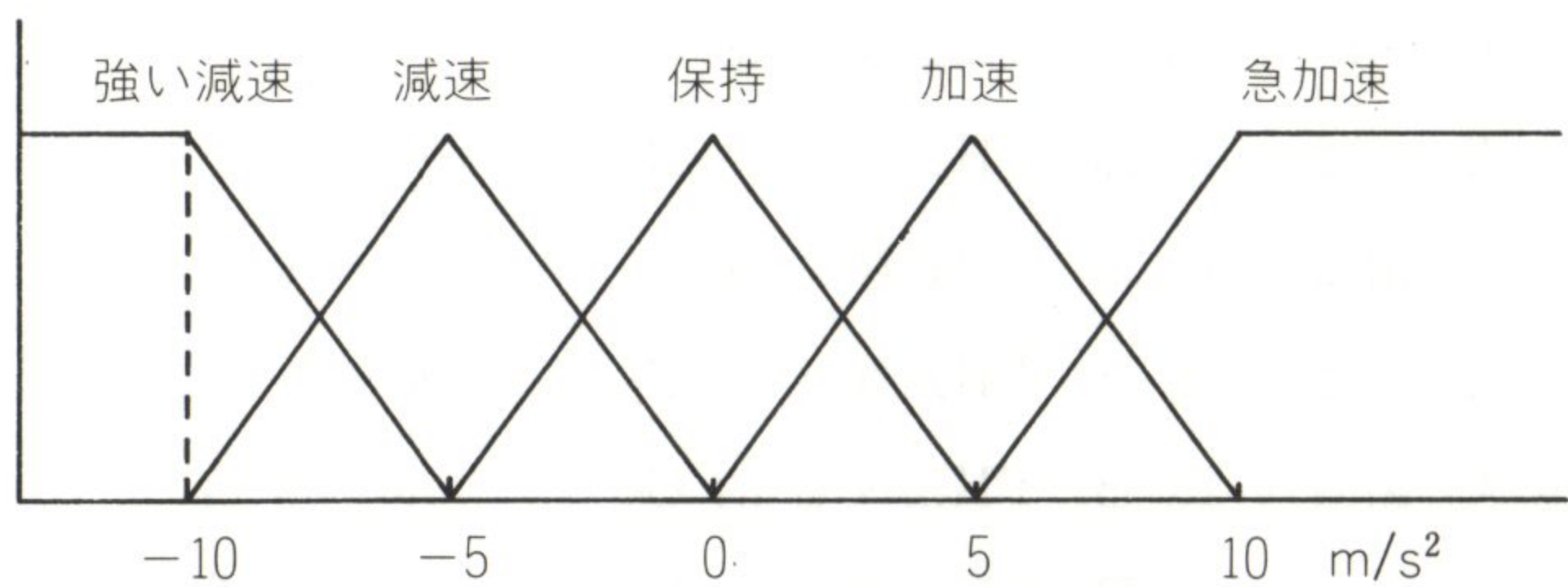
車間の“狭い”，“そこそこ”，“広い”もファジィ集合として，メンバーシップ関数で表すことができます．それを図2.6.3に示します．図から，車間距離20mは，“狭い”のファジィ集合にグレード0.5で，“そこそこ”のファジィ集合にもグレード0.5で属していることがわかります．したがって，車間距離20mは“狭い”とも“そこそこ”な距離であるとも，同じ程度に言えることになります．

現在の車のスピードと車間距離を与えて，スピードのコントロールをするのも

●図2.6.2

●図2.6.3

狭い　そこそこ　広い
1
10　30　50　m

●図2.6.4

強い減速　減速　保持　加速　急加速
−10　−5　0　5　10　m/s²

ファジィ集合で表されます．ここでは，それらを“強い減速”，“減速”，“保持”，“加速”，“急加速”と名づけます．図2.6.4にメンバーシップ関数を示します．減速の最大値を−10Km/s²，加速の最大値を10Km/s²とします．

以上で，スピード，車間，スピードコントロールを表現するファジィ集合——言い換えるならば，あいまいな日常語——を定義してきました．

2.6.2 ファジィ推論

前節で述べた，

スピードが速くて，車間が狭いならば，ブレーキを踏む

はルールと呼ばれます．前提条件のスピードが速いとか車間が狭い，帰結のブレーキを踏む，などはいずれもファジィ集合で表すことができます．一般にファジィ

集合を大文字のアルファベットA，B，C，…，その要素を小文字x，y，zなどで表します．前節の図2.6.2から図2.6.4のファジィ集合を，スピードが“遅い”，“まあまあ”，“速い”をA_1，A_2，A_3，車間の“狭い”，“そこそこ”，“広い”をB_1，B_2，B_3，スピードコントロールの程度“強い減速”，“減速”，“保持”，“加速”，“急加速”をC_1，C_2，C_3，C_4，C_5とします．

ルールの一般的な表現は次のとおりです．

IF　前提部　THEN　帰結部

上の記号を用いて，すべてのルールを表すと次のようになります．

IF x is A_1 and y is B_1 THEN z is C_3　　ルール 1
IF x is A_1 and y is B_2 THEN z is C_4　　ルール 2
IF x is A_1 and y is B_3 THEN z is C_5　　ルール 3
IF x is A_2 and y is B_1 THEN z is C_2　　ルール 4
IF x is A_2 and y is B_2 THEN z is C_3　　ルール 5
IF x is A_2 and y is B_3 THEN z is C_4　　ルール 6
IF x is A_3 and y is B_1 THEN z is C_1　　ルール 7
IF x is A_3 and y is B_2 THEN z is C_2　　ルール 8
IF x is A_3 and y is B_3 THEN z is C_3　　ルール 9

ここで，xはスピード，yは車間，zはスピードコントロールの具体的な値をとります．

これらのルールは，人が任意に定めたものですが，一応常識的な判断からは大きくはずれていないはずです．このルール以外のルールが認められないとか，存在しないとか言うものではありません．

推論は以下のように考えます．

前提部　x is A_i and y is B_j (i,j=1，2，3)

はxが集合A_iに，yが集合B_jに属することを意味します．xとyの値によりその“らしさ”すなわちグレードの値は異なります．xとyはもとの集合，すなわち台集合が異なるのでそのままandをとることはできませんが，xとyの積集合を考えて(x, y) is $A_i \cap B_j$ とします．そして，そのグレードとして，xとyのグレードの小さい

方の値をとることにします．これは，普通の集合演算において，andはその要素が同時に前後の集合の要素である，という演算に相当します．

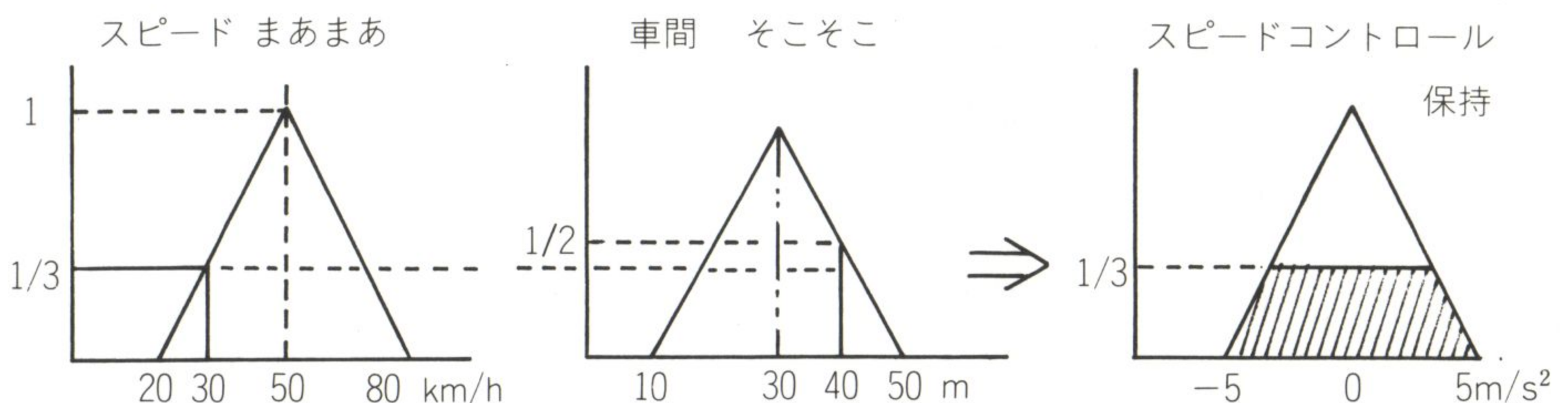
推論は前提部に具体的な条件が与えられたときの帰結部として得られます．マンダーニによれば推論A∩B→CはA，B，Cの積集合で与えられます．すなわち，おのおののグレードの最小値をとることになります．たとえば，ルール５を適用する場合は図2.6.5に見られるようになります．

スピードが30Km/hで，車間が40mであったとすると，ファジィ集合“まあまあ”と“そこそこ”のグレードは1/3と1/2になります．そこで，A∩Bのグレードは1/3です．ファジィ集合“保持”のグレードは1/3以上の値がとれないので，推論された結果のメンバーシップ関数は，図2.6.5の右のグラフのように三角形の頭が切れた台形となります．推論されるのがまたファジィ集合である点に注意してください．また，前提部のAとBの要素のおのおのについて図2.6.5が存在します．すなわち，一般には，３次元空間での話です．

結果を制御に使うためには，ファジィ集合の形では困るので，なんらかの数値化をします．ここでは，ファジィ集合の重心をとることにします．図2.6.5の結果は左右対称なので，計算するまでもなく$0m/s^2$を得ます．

ルールが複数存在する場合には，推論されたファジィ集合の和をとります．ファジィ集合で和をとるとは，グレードの最大値をメンバーシップ関数の値とすることを意味します．

以上のような考えに基づいて，プログラムを作成してみます．

●図2.6.5

2.6.3 プログラミング

前節までに説明したスピードコントロールのモデルをプログラムしてみます．プログラムのポイントは，

- ●メンバーシップ関数の表現
- ●ルールの表現
- ●前提部の処理
- ●重心の計算

の4点です．これらを読みやすく記述できるか否かが，すっきりとしたプログラムが仕上がるかどうかの分かれ道です．

2.6.3.1 メンバーシップ関数の表現

2.6.1で用いたメンバーシップ関数は，3種類の形をしています．それらを図2.6.6(a)，(b)，(c)で表します．

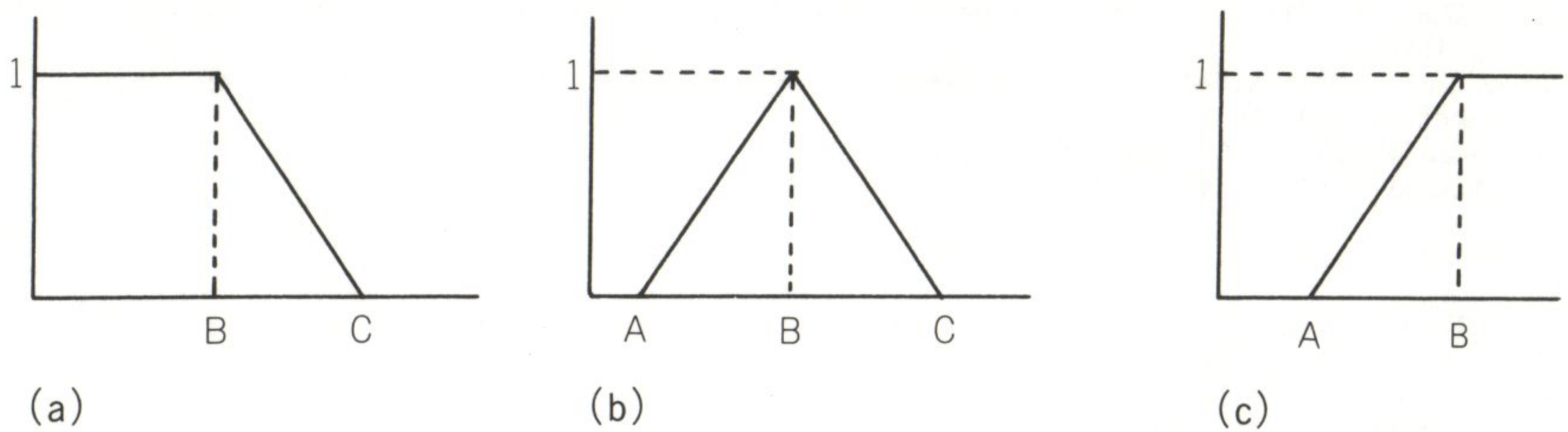
図(b)は，3点A，B，Cを与えれば形が定まります．図(a)は2点BとC，図(c)は2点AとBで形が定まります．これらをひとつのプロシージャにまとめるためには，形を表すパラメータが必要です．ここでは，それを引数のとる値によって区別することにします．図中の点A，B，Cを同じ名前を持つ引数として，

A＝B＜C　ならば　図(a)
A＜B＜C　ならば　図(b)
A＜B＝C　ならば　図(c)

とします．メンバーシップ関数のグレードの最大値は1なので，プロシージャを

```
FUNCTION　grade(A, B, C, x)
```

として，リスト2.6.1のようにプログラムされます．最後の引数xは評価すべき集合の要素の値です．

●図2.6.6

2.6.3.2 ルールの表現

ルールは，前提部が2個のファジィ集合，帰結部が1個のファジィ集合を含みます．おのおのの集合を，**2.6.1**での説明に準じてA_1，A_2，A_3；B_1，B_2，B_3；C_1，C_2，C_3，C_4，C_5とすると，ルールは集合A，B，Cの添字で区別されます．

そこでデータ型RuleStructを

```
TYPE RuleStruct
        Cond1 AS INTEGER
        Cond2 AS INTEGER
        result AS INTEGER
END TYPE
```

として，宣言します．RuleStruct型は，前提部と帰結部の集合の添字をINTEGER型の要素として持ちます．

変数としては，これをルールの数だけの大きさを持つ配列とします．すなわち，

```
DIM Rule(RuleNumber)
```

として宣言します．

2.6.3.3 前提部の処理

前提部は現在のスピードと車間を与えて，グレード値の小さい方をとります．これを，ルールの数だけ実行します．プロシージャEvaluateConditionsとして次のように宣言します．

●リスト 2.6.1 ファジイ制御

```
 1: DECLARE FUNCTION GetCenter! (LowerC, UpperC, C(), Rule() AS ANY, AandB!(), RuleNo!, _
    SpritNo!)
 2: DECLARE SUB EvaluateConditions (VelocityNow!, DistNow!, A!(), B!(), Rule() AS ANY, _
    AandB!(), RuleNo!)
 3: DECLARE FUNCTION grade! (A!, B!, C!, x!)
 4: DECLARE FUNCTION Min! (x!, y!)
 5:
 6: CONST RuleNo = 9               'number of rules
 7:
 8: TYPE RuleStruct                'if x is Cond1 and y is Cond2 then z is Result
 9:     Cond1 AS INTEGER
10:     Cond2 AS INTEGER
11:     result AS INTEGER
12: END TYPE
13:
14: DIM Rule(RuleNo) AS RuleStruct
15: DIM A(3, 2), B(3, 2), C(5, 2), AandB(9)
16:
17: DATA 20,20,50, 20,50,80, 50,80,80: 'set A, velocity of car
18: DATA 10,10,30, 10,30,50, 30,50,50: 'set B, distance between cars
19: DATA -10,-10,-5, -10,-5,0, -5,0,5, 0,5,10, 5,10,10: 'set C, controle of car
20:
21: DATA 1,1,3          : 'Rule 1
22: DATA 1,2,4          : 'Rule 2
23: DATA 1,3,5          : 'Rule 3
24: DATA 2,1,2          : 'Rule 4
25: DATA 2,2,3          : 'Rule 5
26: DATA 2,3,4          : 'Rule 6
27: DATA 3,1,1          : 'Rule 7
28: DATA 3,2,2          : 'Rule 8
29: DATA 3,3,3          : 'Rule 9
30:
31: FOR i = 1 TO 3
32:     FOR j = 0 TO 2: READ A(i, j): NEXT j
33: NEXT i
34:
35: FOR i = 1 TO 3
36:     FOR j = 0 TO 2: READ B(i, j): NEXT j
37: NEXT i
38:
39: FOR i = 1 TO 5
40:     FOR j = 0 TO 2: READ C(i, j): NEXT j
41: NEXT i
42:
43: FOR i = 1 TO RuleNo
44:     READ Rule(i).Cond1: READ Rule(i).Cond2: READ Rule(i).result
45: NEXT i
46:
47: PRINT
48: FOR DistNow = 10 TO 50 STEP 20
49:     PRINT
50:     PRINT USING "Distant = ### M"; DistNow
51:     PRINT "V(Km/h)    Control(m/s^2)"
52:     FOR VelocityNow = 20 TO 80 STEP 10
53:         CALL EvaluateConditions(VelocityNow, DistNow, A(), B(), Rule(), AandB(), RuleNo)
54:         result = GetCenter(-10, 10, C(), Rule(), AandB(), RuleNo, 20)
55:         PRINT USING "#######    ####.####"; VelocityNow; result
56:     NEXT VelocityNow
57: NEXT DistNow
58: END
59:
60:
61:
62: SUB EvaluateConditions (VelocityNow, DistNow, A(), B(), Rule() AS RuleStruct, AandB(), _
    RuleNo)
63:     FOR i = 1 TO RuleNo
```

```
 64:         j = Rule(i).Cond1: k = Rule(i).Cond2
 65:         g1 = grade(A(j, 0), A(j, 1), A(j, 2), VelocityNow)
 66:         g2 = grade(B(k, 0), B(k, 1), B(k, 2), DistNow)
 67:         AandB(i) = Min(g1, g2)
 68:     NEXT i
 69: END SUB
 70:
 71: FUNCTION GetCenter (LowerC, UpperC, C(), Rule() AS RuleStruct, AandB(), RuleNo, SpritNum)
 72:
 73:     LastResult = 0: Area = 0
 74:     FOR x = LowerC TO UpperC + .001 STEP (UpperC - LowerC) / SpritNum
 75:         Maxy = 0
 76:         FOR i = 1 TO RuleNo
 77:             j = Rule(i).result
 78:             y = Min(grade(C(j, 0), C(j, 1), C(j, 2), x), AandB(i))
 79:             IF Maxy < y THEN Maxy = y
 80:         NEXT i
 81:     LastResult = LastResult + x * Maxy: Area = Area + Maxy
 82:     NEXT x
 83:     IF Area <> 0 THEN
 84:         GetCeter = LastResult / Area
 85:     ELSE
 86:         GetCeter = 0
 87:     END IF
 88:
 89: END FUNCTION
 90:
 91: FUNCTION grade (A, B, C, x)
 92:
 93:     IF A = B THEN
 94:         IF x <= B THEN
 95:             grade = 1
 96:         ELSEIF x >= C THEN
 97:             grade = 0
 98:         ELSE
 99:             grade = (C - x) / (C - B)
100:         END IF
101:         EXIT FUNCTION
102:     END IF
103:
104:     IF B = C THEN
105:         IF x >= B THEN
106:             grade = 1
107:         ELSEIF x <= A THEN
108:             grade = 0
109:         ELSE
110:             grade = (x - A) / (B - A)
111:         END IF
112:         EXIT FUNCTION
113:     END IF
114:
115:     IF x <= A OR x >= C THEN
116:         grade = 0
117:     ELSEIF x < B THEN
118:         grade = (x - A) / (B - A)
119:     ELSE
120:         grade = (C - x) / (C - B)
121:     END IF
122:
123: END FUNCTION
124:
125: FUNCTION Min (x, y)
126:     IF x > y THEN
127:         Min = y
128:     ELSE
129:         Min = x
130:     END IF
131: END FUNCTION
132:
```

```
SUB EvaluateConditions(VelocityNow, DistNow, A( ), B( ),
                  Rule( ) AS RuleStruct, AandB( ), RuleNO)
```

引数は次のように働きます.

VelocityNow	:現在のスピードを与える
DistNow	:現在の車間距離を与える
A( )	:集合Aのメンバーシップ関数のパラメータ
B( )	:集合Bのメンバーシップ関数のパラメータ
Rule( )	:ルール
AandB( )	:A∩Bの値(=Min(A, B))
RuleNo	:ルールの数(=9)

具体的なプログラムは**リスト2.6.1**に示されます.

2.6.3.4 重心の計算

スピードと車間が与えられた場合の推論は, 3個の集合の積で与えられます. 今の場合,スピードと車間は1個の値で入力されるので,そのグレードは**2.6.3.3**で述べられたプロシージャEvaluateConditionsにより配列AandB( )に得られています. したがって, ルールごとにスピードコントロールのファジィ集合として得られます(図2.6.5参照). ルールが複数個存在する場合には, ルールごとの推論結果の和(最大値)をとることになります.そのメンバーシップ関数をμ(z)で表すと,重心は,

$$\int z\mu(z)dz \ / \ \int \mu(z)dz$$

で与えられます(積分記号は通常の積分). スピードコントロール値の取り得る範囲は前節で述べたように$-10m/s^2$から$10m/s^2$とします.

プロシージャをGetCenterとして次のように宣言します.

```
FUNCTION GetCenter(LowerC, UpperC, C( ), Rule( ) AS
                  RuleStruct, AandB( ), RuleNo, SpritNum)
```

引数は次のように働きます.

LowerC　：スピードコントロール値の上限
UpperC　：スピードコントロール値の下限
C( )　　：集合Cのメンバーシップ関数のパラメータ
Rule( )　：ルール
AandB( )：A∩B，EvaluatedConditionsで得られた値
RuleNo　：ルールの数
SpritNum：数値的に積分する際の分割数

関数値は重心の値そのものです．

具体的なプログラムはリスト2.6.1に示されます．積分は単純に関数の和を求めています．重心を求めるため分母と分子でΔzは相殺されるのでΔz=1としてあります．

2.6.3.5 プログラム全体

プログラムはメインモジュールレベルコードと，3個のプロシージャからなります．配列A( )とB( )は，スピードと車間のファジィ集合に対する，メンバーシップ関数のパラメータを格納します．

DATA文にて，メンバーシップ関数のパラメータとルールを与えておきます．

結果は，スピードと車間をループ変数にとって表の形にして出力します．

参考文献

本章の執筆に際して参考にした文献を以下にあげておきます．

（1） 寺野，浅居，菅野共編 「ファジィシステム入門」オーム社（1987）
（2） 三矢，田中 「C言語による実用ファジィブック」ラッセル社（1989）

2.7 ファイル送受信プログラム

プログラムやデータなどのファイルを，ほかのパーソナルコンピュータにRS-232C回線を通して送受信する方法を考えてみます．QBのファイル処理機能は従来のBASICのそれよりも強化されているので，プログラミングはその分だけ容易です．

2.7.1 用意すべき機材

パーソナルコンピュータを2台並べて，3.5インチのファイルを5インチのファイルに変換するなどの場合には，俗に“クロス(＝交叉した)”と呼ばれるRS-232Cケーブルが必要です．PC-9801同士の場合には，両方がオスのコネクタでなければなりません．

モデムを使用して，電話で遠くに居る人々へファイルを転送する場合には，モデムとモデムとパソコンの間を接続するケーブルが必要です(モデムを購入すればついてくる)．

概略図を図2.7.1に示します．

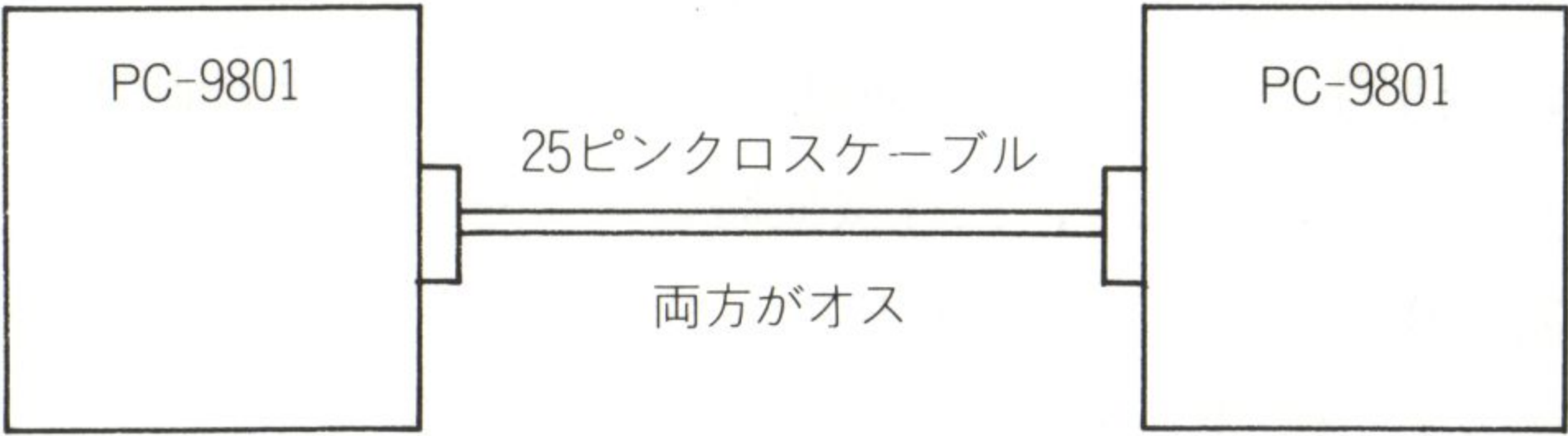
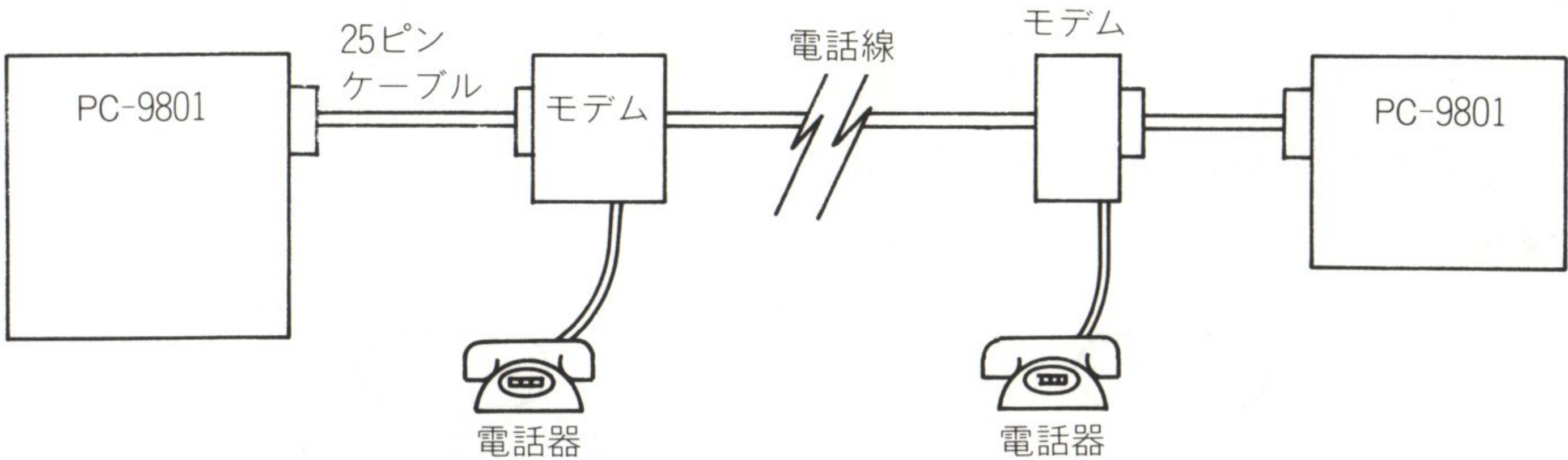
モデムは通常，電話器と回線のモジュラージャックの間に入れて使うようになっています．CCT-98などのパソコン通信用のソフトが動いている場合には，すべて自動で接続されますが，ここでは人がダイアルし相手を確認したうえで(**2.7.4**参照)，スイッチをデータに切り換えて使用します．図2.7.1のような構成になります．

使用するモデムのボーレートは一致している必要があります．ここでは1200ボーのものを用いました．

●図2.7.1

PC-9801
25ピンクロスケーブル
両方がオス
PC-9801

PC-9801同士を直接持続する

PC-9801
25ピン
ケーブル
モデム
電話線
モデム
PC-9801
電話器
電話器

モデムを介して遠くにいる人間と通信をする

2.7.2 短いテキストファイルの送受信プログラム

はじめに，サイズが10Kバイト程度以下の短いテキスト型ファイルの送受信を試みます．テキストファイルに限ったのは，送信終了にEOTコード(＝4)を使用するためです．バイナリファイルですと，すべてのコードがファイル中に含まれるので，次節のように，送信バイト数をあらかじめ送信しておく必要が生じます．

送信自体は別に特別なことではなくて，PRINT文で画面やプリンタにコードを送って文字を印字するのと同じ考えです．ここでは，ファイルをシーケンシャルファイルとしてオープンして，1文字ずつ読み出しては，相手に送信する方法をとります．ファイルにきたらEOTのコードを送信して終了です．受信側は逆に，1文字受け取るごとにファイルにシーケンシャルに書き込んでゆきます．EOTコードがきたら終了です．なお，念のために，受信側は，受信したバイト数を送信側

に返します．送信したバイト数と返事として受け取った受信バイト数が異なった場合には，なんらかのエラーが生じたことを意味します．

全体のフローを送信側，受信側に分けて図2.7.2(a)，(b)に示します．受信側の送信開始信号待ちは，かならずループにして，回線接続時のゴミをキャンセルするようにします．リスト2.7.1(a)，(b)にプログラムリストを示します．

プログラムについて要点を2，3説明します．

◉OPEN　COM

RS-232C回線をオープンするステートメントは，OPEN　COMを用います．N₈₈-BASICと形式は似ていますが，パラメータにボーレートが指定できることや，オプションの置き方が異なっています．

リスト2.7.1(a)，(b)では次のようにオープンしています．

●図2.7.2

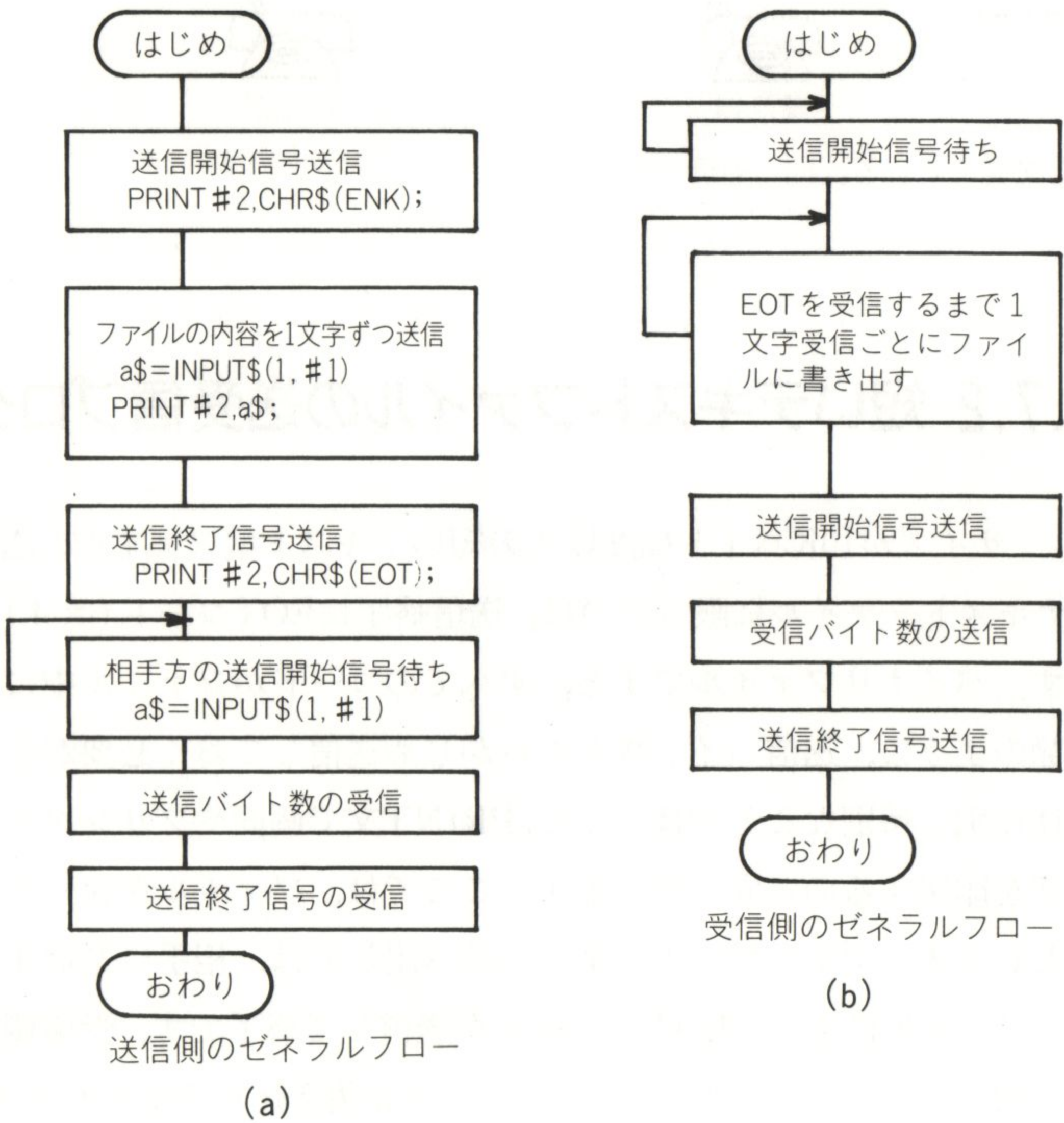
はじめ
送信開始信号送信
PRINT #2,CHR$(ENK);
ファイルの内容を1文字ずつ送信
a$=INPUT$(1,#1)
PRINT#2,a$;
送信終了信号送信
PRINT #2,CHR$(EOT);
相手方の送信開始信号待ち
a$=INPUT$(1,#1)
送信バイト数の受信
送信終了信号の受信
おわり

送信側のゼネラルフロー

(a)

はじめ
送信開始信号待ち
EOTを受信するまで1文字受信ごとにファイルに書き出す
送信開始信号送信
受信バイト数の送信
送信終了信号送信
おわり

受信側のゼネラルフロー

(b)

OPEN "COM1：1200, N, 8, 1, BIN" FOR RANDOM AS #2

最初のパラメータ1200は，1200ボー(1秒間に1200ビットのスピード)で送信することを意味します．このスピードは受信側と一致させる必要があります．モデムを使用する場合には，モデムとも一致させなくてはなりません．次のNはパリティビットのないこと，次の8は，1語8ビットを表します．次の1は，ストップビットが1を表します．最後のBINは，バイナリモードでのオープンを意味します．ここでは，読み出したデータをそのまま送信したいので，BINとします．

FORのあとは，RANDOMとします．このようにすると，入出力いずれにも回線のアクセスが可能となります．INPUTあるいはOUTPUTに指定すると，入力あるいは出力としてしか動作しないので注意してください．

●リスト 2.7.1(a) 送信プログラム

```
 1: CONST EOT = 4   'End of Transmission  伝送終了文字
 2: CONST ENK = 5   'Enquiry              問合せ文字
 3:
 4: filename$ = COMMAND$
 5:
 6: OPEN filename$ FOR INPUT AS #1
 7: OPEN "com1:1200,n,8,1,BIN" FOR RANDOM AS #2
 8:
 9: count = 0
10: PRINT #2, CHR$(ENK);               '送信開始信号
11: DO WHILE NOT EOF(1)
12:     IF count MOD 128 = 0 THEN PRINT ".";
13:     a$ = INPUT$(1, #1)
14:     PRINT #2, a$;
15:     count = count + 1
16: LOOP
17: CLOSE #1
18: PRINT
19: PRINT "送信バイト数 ="; count
20: PRINT #2, CHR$(EOT);               '送信終了信号
21:
22: '相手方の受信バイト数の確認
23:
24: a$ = CHR$(0): DO: a$ = INPUT$(1, #2): LOOP UNTIL a$ = CHR$(ENK)
25: a$ = ""
26: DO
27:   z$ = INPUT$(1, #2)
28:   IF z$ = CHR$(EOT) THEN EXIT DO    '終了信号待ち
29:   a$ = a$ + z$
30: LOOP
31: PRINT "応答バイト数 ="; VAL(a$)
32: CLOSE #2
33:
34: END
35:
```

●リスト 2.7.1 (b) 受信プログラム

```
 1: CONST EOT = 4   'End of Transmission  伝送終了文字
 2: CONST ENK = 5   'Enquiry              問合せ文字
 3:
 4: filenam$ = COMMAND$
 5:
 6: OPEN filename$ FOR OUTPUT AS #1
 7: OPEN "com1:1200,n,8,1,BIN" FOR RANDOM AS #2
 8:
 9: DO
10:     a$ = INPUT$(1, #2)          '送信開始信号待ち
11: LOOP UNTIL a$ = CHR$(ENK)
12: count = 0
13: a$ = INPUT$(1, #2)
14: DO WHILE a$ <> CHR$(EOT)
15:     IF count MOD 128 = 0 THEN PRINT ".";
16:     PRINT #1, a$;
17:     a$ = INPUT$(1, #2)
18:     count = count + 1
19: LOOP
20: CLOSE #1
21: PRINT
22: PRINT "受信バイト数 ="; count
23:
24: FOR i = 1 TO 10000: NEXT i       'delay
25:
26: '相手方へ受信バイト数を送信する
27:
28:     PRINT #2, CHR$(ENK)
29:     PRINT #2, count
30:     PRINT #2, CHR$(EOT)
31:
32: CLOSE #2
33:
34: END
35:
```

◉COMMAND$

QBで作成された実行型ファイル(.EXEのついたファイル)は，コマンド行の文字列を読み込む命令COMMAND$によって，ファイル名などをプログラム起動時に与えることができます．プログラムデバッグ中は，ファイル名filename$には定数で名前を与えておきますが，実行段階では，

filename$＝COMMAND$

とすると，コマンド行で与えられたファイル名を直接プログラムに渡せます．たとえば，送信プログラムのファイル名をsend1.basとすると，これをコンパイルして実行ファイルSEND1.EXEを得ます．そこで，sample.datファイルを転送した

いとすると，

```
SEND1 SAMPLE.DAT ↵
```

のようにタイプすることになります．

◉ENK待ちループ

受信側プログラム(リスト2.7.1(b))のはじめにて，

```
DO
  a$=INPUT$(1, #2)
LOOP  UNTIL  a$  =  CHR$(ENK)
```

とENKコードが送信されるのを待っています．これはモデムを立ち上げたり，相手方のコンピュータ立ち上げの際に，ゴミが送信されることがあるので，本当にENKコードがくるまで待ちます．もっとも，ゴミの中にENKコードと同じ信号が含まれているときにはお手上げですが．

◉送受信バイト数

リスト2.7.1(a)，(b)のプログラムでは，送信したバイト数と受信したバイト数のカウンタは同じ値になります．そこで，それを受信側が送信側に戻し，チェックすることができます．

◉実行方法

はじめに受信側のプログラム2.7.1(b)を実行させ，ついで，送信側プログラム(a)を実行させます．このタイミングが逆になると動きません．その場合には，STOPキーでプログラムを止めてやりなおしてください．

2.7.3 バイナリファイルの送受信プログラム

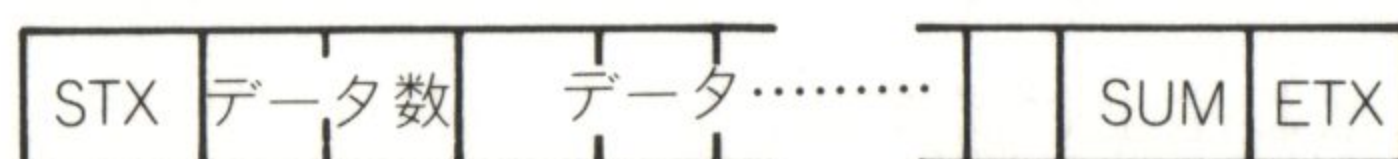
プログラムファイルやバイナリデータを含むファイルには，8ビットのすべての値，すなわち00H～FFHの値が含まれるので，ファイルの終わりにEOTコード(＝4)を送信して，ファイルの終わりを知らせることはできません．そのようなファイルを送受信するためには，送信するデータ数をなんらかの方法で受信側に知らせてやることにします．

考え方としては，次のようにします．すなわち，

(1)　送信する単位をレコードと呼ぶ．

(2)　レコードは次の構造を持つ．

●図2.7.3

STX	データ数	データ………		SUM	ETX

先頭は，STX(＝2)とし，その次にバイナリでデータ数を置く．その順は上位・下位とする．そのあとにデータ数だけのバイト数のデータが続く．データのあとには，データ数も含めた送信コードの合計の下位8ビットの値を置く．

レコードの最後はETX(＝3)とする．

たとえば，データ数を128とすると，レコード全体としては，128＋5＝133バイトの大きさを持つ．

(3)　受信側は，データ数分のデータを受信したら，SUMの値のチェックを行う．正しくなければ，NAK(＝15H，否定応答)を送信側に返す．すると送信側は再送する．再送が5回繰り返されてもエラーが生ずるならば中止する．逆にSUMの値が正しく，かつETXも受信できたら送信側にACK(＝6，肯定応答)を返す．すると，次のレコードを送信してくる．

全体のフローを図2.7.4(a)，(b)に，リストをリスト2.7.2(a)，(b)に示します．前節の例よりいくらか複雑なので，プログラムリストについて補足しておきます．

◉送信側プログラムについて

フローチャート図2.7.4(a)では，プロシージャを使用していないように書いてありますが，1レコード分のデータを，送信用レコードに編成する部分は，SUBプロシージャMakeRecordにまとめてあります．

サムチェック用の値としては，8ビットのみを用いるので，和をとって下位8ビットのみを利用しています．

1回に送信するデータ長を定数BufSize(＝128)で与えます．変数xを，BufSize長の固定長文字列とします．

ファイルからデータを指定したバイト長だけ読み出すには，オープン時に，

```
FOR BINARY
```

とします．こうすると，iを読み出すべきファイルの位置とすると，

```
GET #2,i, x
```

で，xにxのサイズだけのデータが得られます．残りのデータがxのサイズより短い場合には，実際に読み込まれたデータ数はなんらかの方法で算出しなくてはなりません．このプログラムでは，はじめにファイルのバイト数をFileSizeに読み出しておき，そこから算出しています．

xを固定長文字列にしてあるのは，バッファとして使用するためには，そのサイズが一定していることが必要なためです．

ファイル位置をポイントする変数iとファイルサイズを格納しておくFileSizeはLONG型にしておく必要があります．さもないと，32Kバイトより大きなファイルの処理ができなくなります．

◉受信側のプログラムについて

フローチャート図2.7.4(b)で示されるように，1レコード受信部をプロシージャにまとめてあります．レコードの先頭がSTXならば，以下の受信を続けます．EOTであると送信終了コードの受信なので，EOTFlagをtrueにして戻ります．そのほかのコードの場合には，ErrFlagをtrueにします．データを指定されたバイト数だけ受信したあと，サムチェックデータを受信します．それが一致しなければ，ErrFlagをtrueとします．レコードの最後がETXでない場合も同様です．

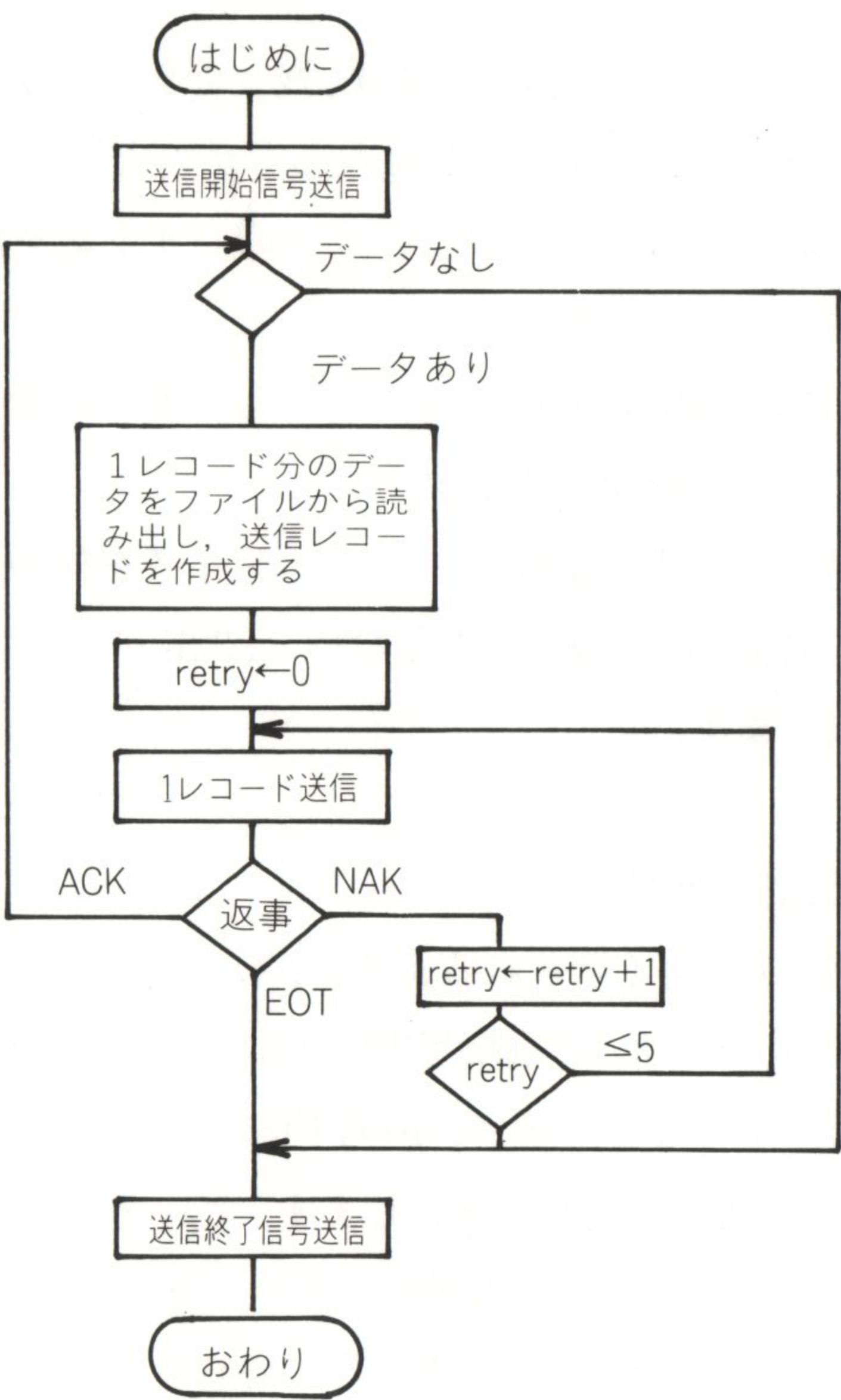
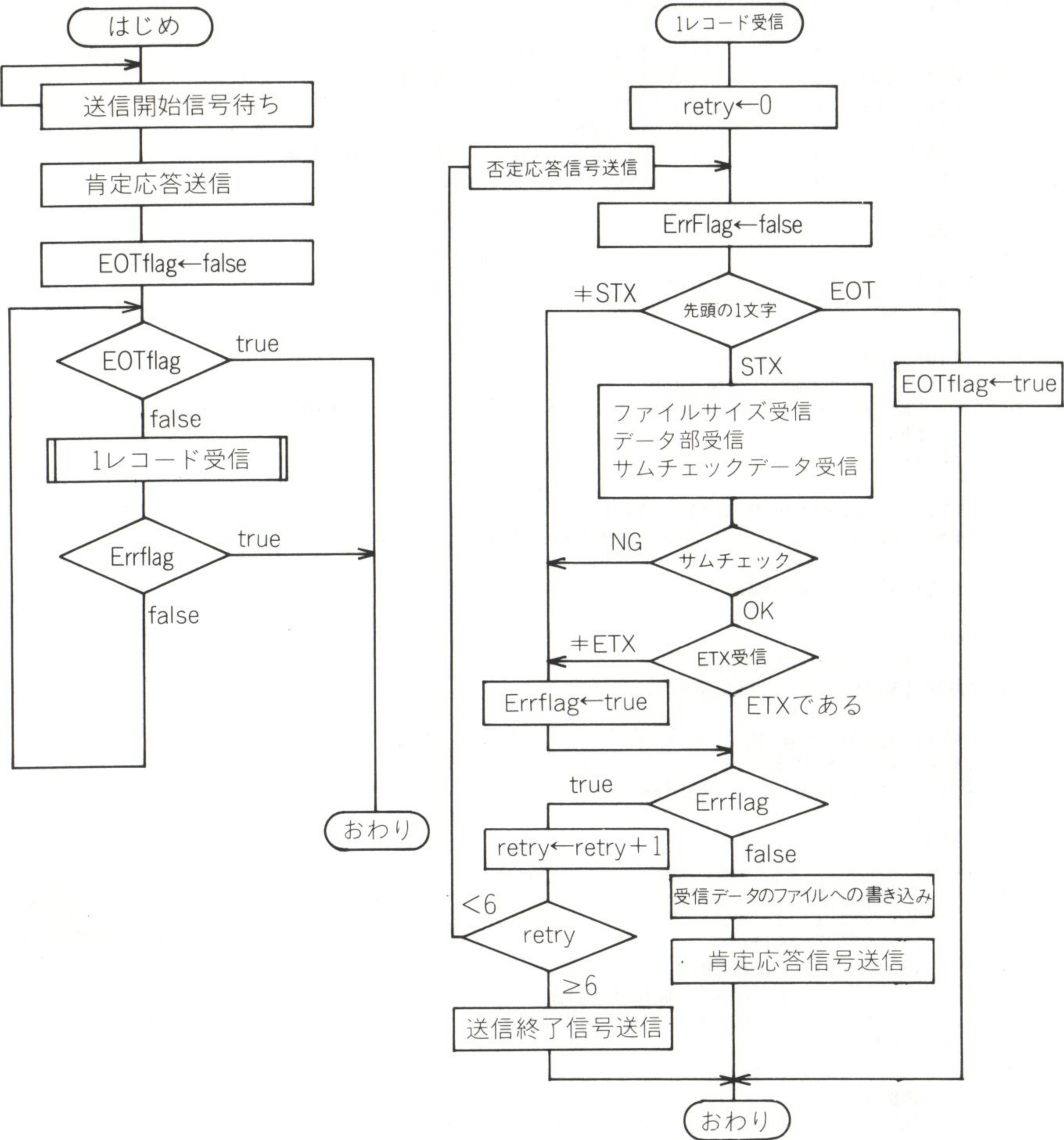
もし，ErrFlagがtrueであると，否定応答文字NAKを送信し，もう一度同一のレコードの送信を要求します．再送が5回連続して実行されてもエラーとなる場合には，送信終了信号を送信して終了します．

EOTFlagとErrFlagの部分に注意すれば，わかりにくい点はないと思います．

受信データを書き込むファイルは，送信の場合と同様に，モードをBINARYとしてオープンします．書き込みは，

```
PUT  #1, i, s$
```

として実行します．s$は送信の場合とは異なり，固定長でなく可変長の文字変数です．これは，少なくとも最終レコードは，レコードサイズがほかのものと異なることが多いからです．ファイルのiバイト目からs$の長さだけデータをファイルに書き込みます．

●図2.7.4 (a) 送信側フローチャート

●図2.7.4（b）受信側フローチャート

●リスト 2.7.2(a) 送信プログラム

```
 1: DECLARE SUB MakeRecord (Size!, s$, t$)
 2:
 3: CONST EOT = 4         'End of Transmission  伝送終了文字
 4: CONST ENQ = 5         'Enquiry              問合せ文字
 5: CONST STX = 2         'Start of Text        テキスト開始文字
 6: CONST ETX = 3         'End of Text          テキスト終了文字
 7: CONST ACK = 6         'Acknowledge          肯定応答文字
 8: CONST NAK = &H15      'Negative acknowledge 否定応答文字
 9:
10: CONST BufSize = 128
11:
12: DIM x AS STRING * BufSize
13: DIM i AS LONG, FileSize AS LONG
14:
15: filename$ = COMMAND$
16: IF LEN(filename$) = 0 THEN
17:     PRINT "ファイルネームが必要です。形式 : send2 filename.exa"
18:     END
19: END IF
20:
21: OPEN filename$ FOR BINARY AS #1
22: OPEN "com1:1200,n,8,1,BIN" FOR RANDOM AS #2
23:
24: PRINT #2, CHR$(ENQ);
25:
26: DO
27:     a$ = INPUT$(1, #2)
28: LOOP UNTIL a$ = CHR$(ACK)
29:
30: PRINT "回線は接続しました。ファイルを送信しています。"
31: FileSize = LOF(1)
32: PRINT "Filesize="; FileSize
33: i = 1
34: PRINT
35: DO
36:     PRINT ".";
37:     GET #1, i, x
38:     i = i + BufSize
39:     IF i > FileSize THEN
40:         Size = FileSize MOD BufSize
41:         IF Size = 0 THEN Size = BufSize
42:     ELSE
43:         Size = BufSize
44:     END IF
45:     CALL MakeRecord(Size, x, t$)
46:
47:     retry = 0
48: again:
49:     FOR j = 1 TO LEN(t$)
50:         z$ = MID$(t$, j, 1)
51:         PRINT RIGHT$("0" + HEX$(ASC(z$)), 2);
52:         PRINT #2, MID$(t$, j, 1);
53:         FOR k = 1 TO 400: NEXT k
54:     NEXT j
55:     a$ = INPUT$(1, #2)
56:     IF a$ = CHR$(EOT) THEN EXIT DO
57:     IF a$ <> CHR$(ACK) THEN
58:         retry = retry + 1
59:         IF retry < 6 THEN GOTO again
60:         EXIT DO
```

```
61:     END IF
62: LOOP UNTIL i > FileSize
63: PRINT #2, CHR$(EOT);
64:
65: CLOSE #1
66: CLOSE #2
67: IF a$ = CHR$(ACK) THEN
68:     PRINT "送信は正常終了しました。"
69: ELSE
70:     PRINT "送信は異常終了しました。"
71: END IF
72:
73: PRINT
74: END
75:
76: SUB MakeRecord (Size, s$, t$)
77:     t$ = CHR$(STX) + CHR$(Size ¥ 256) + CHR$(Size MOD 256) + LEFT$(s$, Size)
78:     Sum = (ASC(MID$(t$, 2, 1)) + ASC(MID$(t$, 3, 1))) AND &HFF
79:     FOR j = 1 TO Size
80:         c = ASC(MID$(s$, j, 1))
81:         Sum = (Sum + c) AND &HFF
82:         PRINT RIGHT$("0" + HEX$(c), 2);
83:     NEXT j
84:     t$ = t$ + CHR$(Sum) + CHR$(ETX)
85: END SUB
86:
```

●リスト 2.7.2(b) 受信プログラム

```
 1: DECLARE SUB RecvRecord (Size!, EOTflag!, ErrFlag!, s$)
 2:
 3: CONST EOT = 4       'End of Transmission  伝送終了文字
 4: CONST ENQ = 5       'Enquiry              問合せ文字
 5: CONST STX = 2       'Start of Text        テキスト開始文字
 6: CONST ETX = 3       'End of Text          テキスト終了文字
 7: CONST ACK = 6       'Acknowledge          肯定応答文字
 8: CONST NAK = &H15    'Negative acknowledge 否定応答文字
 9: CONST False = 0, true = NOT False
10: CONST BufSize = 128
11:
12: DIM x AS STRING * BufSize
13: DIM SHARED i AS LONG
14: DIM FileSize AS LONG
15:
16: filename$ = COMMAND$
17: IF LEN(filename$) = 0 THEN
18:     PRINT "ファイルネームが必要です。 形式：recv2 filename.exa"
19:     END
20: END IF
21:
22: OPEN filename$ FOR BINARY AS #1
23: OPEN "com1:1200,n,8,1,BIN" FOR RANDOM AS #2
24:
25: DO
26:     a$ = INPUT$(1, #2)
27: LOOP UNTIL a$ = CHR$(ENQ)
28: PRINT "回線は接続しました。データを受信します。"
```

```
29:
30: PRINT #2, CHR$(ACK);
31:
32: EOTflag = False
33: i = 1
34: DO WHILE NOT EOTflag
35:     CALL RecvRecord(Size, EOTflag, ErrFlag, s$)
36:     IF ErrFlag THEN EXIT DO
37: LOOP
38:
39: CLOSE #1
40: CLOSE #2
41:
42: IF NOT ErrFlag THEN
43:     PRINT "受信は正常終了しました。"
44: ELSE
45:     PRINT "受信は異常終了しました。"
46: END IF
47:
48: END
49:
50: SUB RecvRecord (Size, EOTflag, ErrFlag, s$)
51:     DIM c AS STRING * 1
52:
53:     retry = 0
54: again:
55:     ErrFlag = False
56:     s$ = ""
57:
58:     c = INPUT$(1, #2)
59:     IF c = CHR$(EOT) THEN
60:         EOTflag = true
61:         EXIT SUB
62:     END IF
63:     IF c <> CHR$(STX) THEN
64:         ErrFlag = true
65:     ELSE
66:         PRINT ".";
67:         c = INPUT$(1, #2)
68:         Size = 256 * ASC(c)
69:         Sum = ASC(c)
70:         c = INPUT$(1, #2)
71:         Size = Size + ASC(c)
72:         Sum = Sum + ASC(c)
73: '        PRINT "RecSize="; Size
74:         cout = 0
75:         DO WHILE count < Size
76: '            IF count MOD 16 = 0 THEN PRINT ".";
77:             c = INPUT$(1, #2)
78:             s$ = s$ + c: count = count + 1
79:             Sum = (Sum + ASC(c)) AND &HFF
80:         LOOP
81:         c = INPUT$(1, #2)
82:         Sum = Sum XOR ASC(c)
83:         IF Sum <> 0 THEN ErrFlag = true
84:         c = INPUT$(1, #2)
85:         IF c <> CHR$(ETX) THEN ErrFlag = true
86:         IF ErrFlag THEN
87:             retry = retry + 1
88:             IF retry < 6 THEN
89:                 PRINT "Retring..."; retry
```

```
 90:            PRINT #2, CHR$(NAK);
 91:            GOTO again
 92:         ELSE
 93:            PRINT #2, CHR$(EOT);
 94:            PRINT CHR$(7); CHR$(7); "Over trials, Aborted."
 95:         END IF
 96:      ELSE
 97:         PUT #1, i, s$
 98:         i = i + Size
 99:         PRINT #2, CHR$(ACK);
100:      END IF
101:   END IF
102: END SUB
103:
```

◉実行方法

送信側のプログラムのファイル名をSEND2.BAS,受信側のそれをRECV2.BASとして，実行ファイルを作成します．はじめに受信側プログラムを，

RECV2 ファイル名 ↵

として起動し，そのあとに送信側のプログラムを，

SEND2 ファイル名 ↵

として起動します．受信側プログラムを起動して，送信側プログラムの起動までの間に時間があきすぎると，うまく動かないことがあります．

2.7.4 電話によるファイル送受信の実例

コンピュータにモデムを接続し，電話を介してファイルを送受信する具体的な例を紹介します．用意するものは，モデムと，コンピュータとモデムを接続するケーブル，およびモデムを電話回線に接続するコードです．これらのコードは，通常，モデムを購入するとついてきます．

接続は，モデムの取り扱い説明書に従って行います．注意すべき点は，こちら側と相手側のモデムのボーレートを合わせることで，これが一致していないと送受信はできません．

2.7.4.1 自動発信のテストー天気予報

現在市販されているモデムはヘイズコマンド(ATコマンド)を採用しています．このコマンドは，文字ATで始まります(例外もある)．ATはAttention(＝注意！)の意味で，モデムの動作確認です．

自動発信させる場合，本来は相手もモデムであるべきですが，モデムのテストとモデムに慣れる意味で，時報とか天気予報などのサービス電話を呼び出してみます．プログラムは次のようになります．

```
OPEN "COM1:1200, N, 8, 1, BIN" FOR RANDOM AS #2
    PRINT #2, "AT"
    DO
        LINE INPUT #2, Z$  :PRINT Z$
    LOOP UNTIL Z$= "OK"
    PRINT #2, "ATD177"
    DO
        LINE INPUT #2, Z$  :PRINT Z$
    LOOP UNTIL Z$= "NO CARRIER"
    CLOSE #2
END
```

はじめに，コマンド“AT”をモデムに送ります．モデムが動作可能ならば，返事(リザルトコードと呼ぶ)“OK”を返してきます(リザルトコードが数字で返ってくるような場合には，コマンドVを調べてください)．それを確認したあとに，ATに続けてコマンドDと3桁の番号177を送ります．すると，自動的にダイアルされて相手が出ます．

モデムとしては相手がモデムだと考えて，接続を保持しようとしますが，実際は天気予報の音声なので，約30秒で接続保持をあきらめ，キャリア信号が見つからないという意味の“NO CARRIER”という返事を返して接続を切ります．

使用する電話が，ダイアル方式かプッシュホン方式かで，上のプログラムが動かないことがあります．それを区別するには，ダイアルコマンドを，

ATDP177… パルスダイアル(ダイアル式)

ATDT177… トーンダイアル(プッシュホン式，“ピッポッパ”のタイプ)

とパラメータPまたはTを付加します．

また，電話が0発信などの場合，0を回してから外線とつながる間に時間がかかります．そのときには，パラメータ“,”を入れると，3秒ほどのポーズがあきます．すなわち，

ATDP0,,177 …パルスダイアルで，0を回して，約6秒してから177をダイアルする

のようにします．

2.7.4.2 自動着信

外部からかかってきた電話に自動的に応答するためには，モデム内部のレジスタS0の値をゼロでない数値にセットします．ここでは，S0の値を2とします．すると，着信時にベルが2度鳴ると，受話器を取り上げたと同様の動作(オフフック)をし，相手側にキャリア信号を送り出します．プログラムは次のとおりです．

```
PRINT #2,"ATS0=2"
```

S0＝2はレジスタS0に値2をセットすることを意味します．

2.7.4.3 ファイル転送の実際

2.7.3で作成されたファイル送信用プログラムSEND2.EXEと受信用プログラムRECV2.EXEを利用して，モデムを介してファイルの送受信を行う簡単なシステムを構成してみます．次のような手順で実行することを想定します．

(1) 人が電話をかける．ファイル送信側と受信側を打ち合わせる．コンピュータとモデムの電源を入れる．いったん電話を切る．

(2) 送信側の手順は次のとおり．
送信側セットプログラムSSET. EXEを起動する．

A＞SSET ⏎

ベルが２度ほど鳴り，そのあと，FAXの接続時に似たピーという音がしたあとに，モデム同士が接続する．

メッセージ

Type　" SEND2　FILENAME"

が現れたら20秒ほど待ってから，SEND2.EXEを起動する．すなわち，

＞SEND2　ファイル名　⏎

終了メッセージが表示されたらモデムの電源を落とす．

（２）'　受信側の手順は次のとおり．

受信側のセットプログラムRSET. EXEを起動する．

A＞RSET ⏎

電話のダイアル音がして，ついで呼び出し音が２度ほど鳴ると，FAXの接続時に似たピーという音がしたあとに，モデム同士が接続される．続いて2,3のメッセージが表示され，最後に，

Type　" RECV2　FILENAME"

と表示される．そうしたら，すみやかに，

A＞RECV2　ファイル名　⏎

とタイプする．

相手がSEND2プログラムを実行してからRECV2が実行されると，ファイルの転送は行われないので，10秒以上の時間をあけないようにすること．

終了メッセージが表示されたらモデムの電源を落とす．

（３）　電話をかけて，ファイルの送受信を確認する．

以上の手順でファイルの送受信を行うためのプログラム，SSET. BASとRSET.

BASのリストをリスト2.7.3(a)，(b)に示します．コンパイルして．EXEファイルとして使用します．

電話回線の不調などで通信がうまく行われなく，コンピュータが何も受けつけない状態になったら，モデムの電源を切り，RS-232C用コネクタを外してください．プログラムは，

デバイスからの応答がありません

などのメッセージを出して終了します．

◉プログラムの改良について

ここで作成したRSET. BASとSSET. BASは，それぞれRECV2.BASとSEND2.BASと一体にして使用することもできます．変更は容易なので，プログラムリストを載せることはしません．

また，コマンドを研究すれば，終了後自動的に電話に戻すことも可能です．それらは読者の皆さんの課題としておきます．

●リスト 2.7.3(a) 着信セットプログラム

```
 1: CONST ACK = 6
 2: CONST CR = 13
 3: CONST false = 0, true = NOT false
 4:
 5: CLS
 6: OPEN "com1:1200,n,8,1,BIN" FOR RANDOM AS #2
 7:
 8: PRINT #2, "ATS0=2"          'set Answer Mode and wait twice call
 9:
10: DO
11:     a$ = INPUT$(1, #2): PRINT a$;
12: LOOP UNTIL a$ = CHR$(ACK)
13:
14:
15: PRINT "Connected..."
16: FOR i = 1 TO 1000: NEXT i
17: PRINT #2, "受信の用意をして下さい。"
18: FOR i = 1 TO 1000: NEXT i
19: PRINT #2, "送信を開始します。"
20: FOR i = 1 TO 1000: NEXT i
21: PRINT #2, "...."
22: PRINT #2,
23: PRINT "Type 'SEND2 FILENAME'"
24:
25: CLOSE #2
26: END
27:
28:
```

●リスト 2.7.3(b) 送信セットプログラム

```
 1: CONST ACK = 6
 2: CONST false = 0, true = NOT false
 3: CLS
 4: OPEN "com1:1200,n,8,1,BIN" FOR RANDOM AS #2
 5:
 6: PRINT #2, "ATDxxxxxxx"        'set Call Mode  xxxxxxxの 部 分 に 電 話 番 号 を 入 れ る
 7: COM(1) ON
 8: ON COM(1) GOSUB RecvMsg
 9:
10: StartFlag = false
11: DO
12: LOOP UNTIL StartFlag = true OR AbortFlag = true
13:
14: IF StartFlag THEN
15:     PRINT "Please type 'RECV2 FILENAME'"
16: ELSE
17:     PRINT "Disconnected."
18: END IF
19:
20: CLOSE #2
21: END
22:
23: RecvMsg:
24:     COM(1) OFF
25:     LINE INPUT #2, a$
26:     IF INSTR(a$, "...") THEN StartFlag = true
27:     IF INSTR(a$, "NO CARRIER") THEN AbortFlag = true
28:     IF INSTR(a$, "CONNECT") THEN
29:         PRINT ":"; a$
30:         FOR i = 1 TO 10000: NEXT i:
31:         PRINT "Now send ACK code.": PRINT #2, CHR$(ACK);
32:     ELSE
33:        PRINT ": "; a$
34:     END IF
35:     IF NOT StartFlag THEN COM(1) ON
36:     RETURN
37:
38:
```

2.7.5 注意事項

(1) 本章では，ボーレートを1200ボーにしてありますが，コンピュータによっては，処理が間に合わない場合があります．その場合には，送信側プログラムの１文字送信のあとにおかれている，ダミーのFORループのループ回数を多くしてください(リスト2.7.2(a)の下の部分)．

```
52:  PRINT  #2, MID$(t$, j, 1);
53:  FOR  k=1  TO  1  :NEXT  k     ←このループを加減する
```

(2) QB以外の言語では，ボーレートの設定をプログラム中から行えない場合もあります．また，他機種のコンピュータでは，ボーレートや語長などを与えるパラメータの形式が異なっていることがあります．その点に注意してください．ここでは，

ボーレート	1200ボー
語長	8ビット
パリティビット	なし
ストップビット	1

となっています．

[付録] 制御コード(コントロールコード)について

情報交換用の文字コードのうち，00から1FHの32個は非図形文字コードで，制御コードと呼ばれています．それぞれに名前がつけられており，7はBELであって，これをプリンタやディスプレイに送ると，ビープ音が鳴ります．13はCR，10はLFなどはよく知られています．本章で用いたコードは，そのほかにACK, EOT, NAKなどがあります．記号と名称を次ページにまとめておきます．

キャラクタ	コード	英　語	定　義	意　味
ACK	06H	Acknowledge	肯定応答	送信側に対する肯定的な応答
BELL	07H	Bell	ベル	注意を喚起する
BS	08H	Backspace	後退	1キャラクタ分後退させる
CAN	18H	Cancel	取消	先行データの取消
CR	0DH	Carrige Return	復帰	同一行の先頭に戻す
DEL	7FH	Delete	抹消	誤りの除去
DLE	10H	Date Link Escape	伝送制御拡張	伝送制御機能の拡張を指示
ENQ	05H	Enquiry	問い合わせ	相手の応答の要求
EOT	04H	End of Transmission	伝送終了	テキストの終了を表す
ESC	1BH	Escape	拡張	制御機能の拡張に用いる
ETB	17H	End of Transmission Block	伝送ブロック終了	分割されたブロックの終結を示す
ETX	03H	End of Text	テキスト終結	一個のテキストの終結を示す
NAK	15H	Negative Acknowledge	否定応答	送信側に対する否定的な応答
SOH	01H	Start of Heading	ヘッディング開始	ヘッディングの最初のキャラクタとして用いる
STX	02H	Start of Text	テキスト開始	テキストの最初のキャラクタ，ヘッディングのある場合はその終結
SYN	16H	Synchronous Idle	同期信号	他のキャラクタを伝送しない場合に同期を取るために使用する

索 引

Quick BASIC

記号／英数字項目

アルファベット順

和文項目

あ

か

さ

た

な

は

ま

や／ら／わ

■著者略歴
小池慎一（こいけしんいち）
1969年 名古屋工業大学大学院修士課程修了
現在名古屋文理短期大学情報処理学科助教授
主な著書に「改訂第2版 はじめてのPascal」，
「Mathmatica 数式処理入門」(技術評論社)，
「Cによる科学技術計算」,「連続系シミュレーション」
(CQ出版社)．訳書に「UCSDpシステム入門」，
「アンダースタンディングC」(CQ出版社)，
「BASIC言語比較研究」(技術評論社)などがある．

森　博（もりひろし）
1979年 名古屋工業大学大学院修士課程修了
現在名古屋文理短期大学情報処理学科助教授
著書に「情報処理システム入門－操作編－(共著)」
(中部日本教育文化会)がある．

Quick BASIC中級プログラミング

平成2年10月25日　初版 第1刷発行
平成3年8月15日　初版 第2刷発行
著　者　小池慎一・森　博
発行者　片岡　巖
発行所　株式会社技術評論社
　　　　東京都新宿区愛住町8番地8
　　　　電話　03(3225)2300　営業部
　　　　　　　03(3225)3293　編集部
印刷／製本　日経印刷株式会社

定価はカバーに表示してあります

ISBN4-87408-387-0 C3055

Printed in Japan

M-cord 210071

Quick BASIC
中級プログラミング

小池慎一＋森 博 著

ISBN4-87408-387-0 C3055 P2500E 定価2500円（本体2427円）